ADVANCING RELIABILITY, MAINTAINABILITY, AND AVAILABILITY ANALYSIS THROUGH A ROBUST SIMULATION ENVIRONMENT

by

Stephen Paul Chambal

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

May 1999

ADVANCING RELIABILITY, MAINTAINABILITY, AND AVAILABILITY ANALYSIS THROUGH A ROBUST SIMULATION ENVIRONMENT

by

Stephen Paul Chambal

has been approved

April 1999

APPROVED:

J. Bert Keats ,Co-Chair

Donald T. Mackulak ,Co-Chair

Supervisory Committee

ACCEPTED:

Department Chair

Dean, Graduate College

ABSTRACT

The field of reliability has become a dominant factor in industry as companies strive for competitive advantages in the market place. Reliability, maintainability, and availability (RM&A) analysis provides insight into system characteristics and overall behavior, describing parameters such as time between failures, mean down time, and cost estimation. For simple systems, analytical methods can calculate exact solutions. Complex systems, however, require different methods for approximating solutions. Simulation is one solution for evaluating system performance and advancing the understanding of RM&A metrics.

The first phase of this research focuses on how data requirements for input can affect simulation results. When failure data is sparse, the triangle distribution can replace the Weibull distribution to describe component failure rates. This reduces the complexity of the modeling process while returning similar output analysis. The research results indicate that the availability metric for system performance is insensitive to input distribution specification. The input parameters for the triangle distribution, which replace the true Weibull failure distribution, can be varied as much as 30% with little impact on system availability. With respect to mean time to first failure, it appears that overestimating the triangle mode and underestimating the worst case failure (maximum parameter) provides accurate prediction of system performance. Overall, the triangle distribution offers a valuable alternative to the Weibull distribution when little or no data is available.

The next phase of this research uses simulation results as the basis for comparing sparing alternatives when evaluating competing systems. Sparing strategies are

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	2.Jul.99	DISSERTATION

4. TITLE AND SUBTITLE
ADVANCING RELIABILITY, MAINTAINABILITY, AND AVAILABILITY ANALYSIS THROUGH A ROBUST SIMULATION ENVIRONMENT

5. FUNDING NUMBERS

6. AUTHOR(S)
CAPT CHAMBAL STEPHEN P

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ARIZONA STATE UNIVERSITY

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
FY99-135

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

14. SUBJECT TERMS	15. NUMBER OF PAGES	16. PRICE CODE
	241	

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

compared on the basis of availability and cost. Initial simulation results provide three alternatives for supporting a simple communications network. The present value of money is incorporated into the analysis due to the component reliabilities and time the system survives. The decision-maker is provided a selection rule based on the cost factors and simulated failure times for this scenario. The results are specific to the provided case study, but the method followed for completing the analysis can be applied to any system.

The final phase of this research focuses on the time between failure and time to failure distributions. The time between failure is exponentially distributed but the time to failure is not. A sparing example clarifies the two distributions and highlights the errors associated with misapplying the exponential distribution to the time to failure density. Excessive sparing is required, and the system unfairly experiences a low availability. Using the proper distribution and parameters may prevent a company from discarding a good product design or allocating unneeded resources to sparing inventory. The intent of this study is to clarify the confusion present in the reliability literature.

DEDICATION

This research is dedicated to the one I love, my beautiful wife. You are the reason for my continued success. I could not have finished this research without your motivation, guidance, editing, and persistent motivation. As always, you are stuck with me for the rest of your life; I am the lucky one. Special thanks to my brother and his family for putting up with another Arizona tour. I'm sure you will miss my 7:00 a.m. wake up calls and late night cribbage sessions. Our time here would not have been so enjoyable without you and Sponkadoo. We'll see you when the snow comes.

ACKOWLEDGEMENTS

I would like to take this opportunity to express my deep gratitude and appreciation for the members of my committee. Dr. J. Bert Keats, it was your encouragement and recommendation to the Air Force that made this program a reality. Your support continued throughout my entire program, thank you. Dr. Mackulak, you provided a constant reminder to demonstrate a connection between my research and the real world. I would like to thank you and Dr. Hogg for accepting my invitation to sit on my committee.

It is nice to know that you are not alone. I sincerely thank my fellow students for their advice and assistance throughout this ordeal. A special thanks to Sharon Lewis for showing me true dedication to research and a willingness to help others; good luck in England. Dr. Lanning, I look forward to teaching with you in Ohio as our parallel lives continue. Jim Wisnowski, I respect your advice and admire your constant focus on what lies ahead. We endured this process together and I hope our friendship continues as we depart on our own paths.

I would like to thank the Arizona State University faculty and staff for assisting me in this adventure. The following people kept my program on track and made my interaction with the department a great experience: Beverly Naig, Charleen Smith, Joyce Potterf, and Barbie Orcutt. Finally, I would like to thank the Air Force Institute of Technology for providing me an opportunity to continue my education.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Background

The United States Air Force is very active in developing new techniques to evaluate military weapons systems and support equipment. The Air Force Operational Test and Evaluation Center (AFOTEC) is responsible for planning and conducting Operational Test and Evaluation (OT&E) for new systems. Their mission is to determine system operational effectiveness and suitability, objectively and impartially (AFOTEC Mission Statement). Suitability refers to the "ilities" of equipment performance, i.e., reliability, maintainability, and availability (RM&A). AFOTEC is responsible for testing new equipment to ensure the RM&A specifications purposed by the contractor during acquisition are achieved.

There are many circumstances that prevent full testing of equipment prior to acquisition by the Air Force. There are time and cost constraints that can not be ignored. A satellite system with a 10-year reliability requirement can not be tested to verify the performance specifications within months of an OT&E scenario. Weapon systems are too expensive to determine lethality and accuracy through live fire or destructive testing. These are just two examples of limiting factors driving the Air Force to create alternative methods system evaluation. Simulation is one option for predicting system performance which is being pursued by the Air Force.

The Studies and Logistics office of AFOTEC began developing a reliability software package to be used for evaluating system performance. The simulation software

designed was generic in nature and driven through a graphical user interface. The end product is RAPTOR, Rapid Availability Prototyping for Testing Operational Readiness. The software package is used by AFOTEC and other testing agencies throughout the Air Force. The goal of RAPTOR is to enhance existing RM&A analysis capabilities and provide a generic modeling capability to evaluate proposed systems for acquisition (RAPTOR User's Manual (1995)). Additional details in Chapter 2 explain the operation and performance of RAPTOR.

This research stems directly from RAPTOR and my involvement with this software. I was a member of the Studies and Analysis division of AFOTEC and part of the test team for beta-testing the reliability software. The approval of RAPTOR led to its release in the public domain for the industrial and academic community. The Air Force began using RAPTOR for satisfying testing requirements and responding to questions on system suitability and effectiveness. The industrial and academic community began using RAPTOR in a similar fashion as an alternative approach to system analysis. My involvement with Arizona State University led to this research, connecting the military version of RAPTOR with the industrial community backed by continued academic research.

Research Motivation

Reliability is quickly becoming a dominant factor for product and process improvement. A company improves its competitiveness by increasing product reliability, maintainability and availability. The critical time to implement changes to product and process design is during the developmental stages on new products. Having the

capability to predict system performance increases the company's chance to make modifications to equipment in order to gain profitability after development. Reliability simulation offers an avenue for early prediction of system performance. Reliability analysis is used to compare design alternatives before production and choose the best option along with identifying and removing problems at the same time (Moss (1996)). Furthermore, increasing the understanding of RM&A through simulation enhances management's ability to make intelligent and well-informed decisions.

A company capable of accurately predicting system performance gains an edge in research and development of new products. Resources can be applied to the critical areas of design. Reliability improvements can be focused in areas that provide the biggest incentive for increasing system capability. Logistic issues can be addressed prior to having the equipment in place and deployed. Analyst can perform "what if analysis" to determine benefits of alternative maintenance strategies and understand the benefits of implementing one system versus another. Providing this type of capability to the reliability analyst in the industrial community is significant.

The key to answering RM&A concerns is to develop a method for conducting detailed analysis beyond the capability of analytical solutions. Furthermore, the solution must be easy to implement by the practicing engineer with limited exposure to simulation programming and coding structures. The goal is to advance the understanding of RM&A issues and offer the analyst a simple method for conducting research through generic simulation. The opportunity has presented itself by incorporating the RAPTOR simulation software into advanced research in the academic arena. The software is

available in the public domain and thus removes any obstacle of obtaining an executable version of the program.

My unique involvement with the Air Force Operational Test and Evaluation center has allowed me to access the source code for the simulation tool. The supporting documentation has also been provided and arrangements have been made to accommodate changing the coding structure and features available in RAPTOR. The Air Force approves of joint research and has given the support of additional software and resource assistance when required. The opportunity to pursue advancements in the field of RM&A is furthered by the involvement of Arizona State University and the neighboring industrial community.

Problem Statement

The focus of this research is to address reliability, maintainability, and availability concerns from the angle of the practicing engineer. The direction of improvements must be made to allow the routine analyst to understand and replicate the findings as applied to their own, individualized problems. The problem is taking a complex and confusing situation and providing an easy to follow algorithm for handling RM&A analysis. The availability of a generic modeling tool is combined with this research to enhance the fundamental contribution to the reliability community.

There are many existing definitions for reliability, maintainability, and availability in the field of reliability engineering. Kapur and Lamberson define system reliability as "the probability that, when operating under stated environmental conditions, the system will perform its intended function adequately for a specified time".

Furthermore, "maintainability is defined as the probability that a failed system can be made operable in a specified interval of downtime", and "availability is defined as the probability that a system is operating satisfactorily at any point in time . . .", and "is a measure of the ratio of the operating time of the system to the operating time of the system plus the down time" (1977).

System level designs are configured using reliability block diagrams (RBDs) to complete RM&A analysis. A reliability block diagrams is a "graphical and mathematical representation of the relationship between a system's components (blocks) and their effect on the resulting system readiness level" (RAPTOR User's Manual (1995)). Reliability block diagrams integrate individual component behavior into system performance criteria. The RBDs include a combination of operating blocks, nodes and links connecting the overall design to describe the structure of the system. A simplistic RBD includes both parallel and series components as shown below:

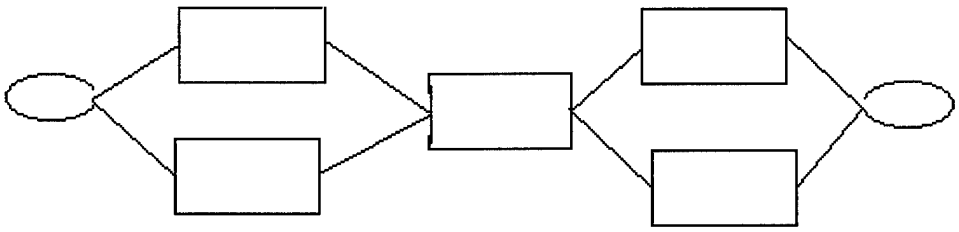


FIGURE 1-1. Simple Reliability Block Diagram

The system includes multiple input and output paths, and the blocks are connected based on the interaction of components in the physical system. The reliability block diagram could represent individual components or a set of a subsystem configured to represent the behavior of the system as a whole.

The series components must be working for the given path to be operational. The parallel components, on the other hand, must have "k out of n" working paths to have the system operational. The links and the nodes simply connect the entire system providing an image of the structure and how the components are related. This RBD represents a simple example, and the structure quickly becomes complex as the reality of a real-world system is designed. Analytical methods are used to analyze the most basic RBDs and this option is discussed in more detail in Chapter 2.

Realistic situations force the analyst from an analytical solution to choosing an alternative method to predict system performance. Reliability simulation is one alternative and accurate modeling can improve analyst predictions. The key is to include enough detail to model the real-world system without causing too much additional burden on the analyst. The generic modeling approach allows the user whose is unfamiliar with simulation to build basic models and describe system behavior. This continues to be an area of concentration throughout the research.

The fundamental issue is two-fold; the current level of RM&A analysis needs to be expanded to answer questions on reliability assessment, and secondly, misunderstood analysis must be clarified to prevent inaccurate use by reliability engineers. Expanding the current level of understanding in reliability analysis provides a unique contribution

for engineers. By identifying new analysis and providing examples of applied scenarios, analysts are more informed and capable of performing assigned tasks. Clarifying areas of confusion in reliability analysis prevents the engineer from misapplying information and drawing erroneous conclusions based on derived results.

The primary avenue for expanding and clarifying reliability issues is through simulation. A secondary problem analyst face is the skill and education required for programming a simulation model for complex systems. Many reliability engineers are skilled in the area of RM&A and applied statistics, but are not code writing analysts. Therefore, the issue is creating an environment that allows the easy conversion from the reliability world to the simulation software. The option pursued in this research uses reliability block diagrams as building blocks in a generic modeling tool. Benefits of generic modeling and an overview of existing software tools are covered in Chapter 2.

The situation now becomes data dependent. Input data is required to generate an accurate model and calculate system statistics. The analyst must maintain a wide range of information to build the appropriate reliability block diagram. Individual component behavior affects the outcome of simulations. The failure rates, repair rates, and logistics information are examples of input data required. Further confusion is generated when little or no data is available to build the appropriate failure and repair distributions. The output of system statistics is affected by the quality of input data. The problem is one of sensitivity and the effect on modeling accuracy as input quality decreases. All simulation models are data driven; this problem can not be ignored.

Assuming the issue of data is addressed, the engineer is concerned with model accuracy. The realistic aspects of real-world problems must be incorporated into the simulation program. Many software applications are too general and do not account for details affecting system performance. Limitations are built around distribution selection, maintenance strategies and logistics information. Simplifying assumptions forced on the user can negate any accuracy in output parameters. The reliability community needs a software tool that includes an adequate amount of detail to give credibility to simulated results. Without realistic modeling capability, simulation is not effective as a decision-making aid.

Time and resources also become an issue when faced with finding alternative methods to perform reliability analysis. Time is always an issue. Writing simulation models from the ground up requires a tremendous amount of time and resource, along with the skilled and knowledge requirements mentioned earlier. Companies continue to want answers to reliability questions in a matter of days, hours, and even minutes. The speed at which results can be obtained is dependent on the nature of the solution. An ideal solution converts a complex system into an understandable format, and provides immediate results on system characteristics. Advancements in reliability, availability and maintainability analysis needs to account for time and resource allocation. The lower the cost and quicker the tool, the higher the acceptance in the industrial setting.

Research Goals

System level reliability, maintainability and availability (RM&A) analysis supports product and process improvement through increased understanding of system

performance. The Department of Defense and the United States Air Force rely heavily on RM&A analysis during test and evaluation in the acquisition process of new weapons and support equipment. Reduced budget and limited manpower requires alternative decision-making tools when evaluating the effectiveness and suitability of these new systems. The industrial community further supports RM&A analysis as a means to improve products and processes to remain competitive in a global economy.

Complexity issues with maintenance practices, component behavior and other areas quickly force analysts away from analytical solutions. Limitations on data collection and component information further hinder efforts to accurately predict system behavior. Although simulation offers possible solutions, coding knowledge and excessive programming time cause difficulties in implementing a modeling solution. Also, the RM&A practitioner may not have the programming skills required to effectively model the system behavior. Modeling enough detail to simulate real-world characteristics affects the accuracy of the results when they are obtained.

The purpose of this research is to advance the understanding of RM&A analysis in four specific areas. First, allow engineers to solve for performance metrics with little or no data combined with testing the metric sensitivity to changes in the input parameters. Specifically, a triangle distribution replaces the component's Weibull failure density, and availability and reliability metrics are calculated. The results are compared and conclusions are drawn with respect to modeling RM&A criteria with sparse data.

Secondly, the research tries to incorporate realistic maintenance policies into RM&A evaluations in order to produce realistic models predicting system performance.

The changes for handling additional maintenance features can not require additional coding and simulation skills. The goal is to maintain easy to use software tools to remove the burden of programming from the practitioner. Furthermore, the benefits of enhancing modeling features are demonstrated using applied examples. The time and resource constraints are minimized while providing accurate solutions for problems facing reliability engineers.

Next, an overlying cost structure is developed and applied to system RM&A analysis. Cost predictions support management-level decision making and provide a metric for comparing alternative reliability policies. The cost structure is related to maintainability and operation of the system. Both areas are considered important and are accounted for during the development of new techniques for handling RM&A analysis. The goals of the following research are to expand the level of knowledge in understanding reliability assessment of complex systems. The tools are provided as a means for advancing RM&A analysis and doing so with limited skill in the area of simulation.

Finally, this research clarifies reliability results to prevent erroneous conclusions based on misunderstood literature. Time between failure (TBF) and time to failure (TTF) are often confused and applied in the wrong situations. The goal is to demonstrate the correct application of past theoretical contributions and highlight the penalties for incorrect analysis. Many engineers make decisions based on misinterpreted results and the confusion in applying the correct TBF or TTF distributions. The results are over expenditure for sparing and logistics and a waste of company resources. Saving capital

by preventing inaccurate maintenance predictions is advantageous to improving a company's performance.

Expanding and clarifying reliability analysis centers around the development of a robust simulation environment. The modeling tools are reviewed and Monte Carlo simulation is offered as a solution for completing this research. The final decision is to improve existing state-of-the-art software through joint research with ongoing Air Force projects. The Air Force Operational Test and Evaluation Center (AFOTEC) provides support and assistance for this modeling research and feedback is accepted for suggested improvements. The joint nature of this research allows for immediate integration of RM&A contributions into emerging versions of reliability software.

Importance of the Study

Academic approaches to solving reliability issues are limited in terms of analytical solutions. Only simple systems meet the required assumptions to fit analytical equations and lead to questionable results. Lifting the assumptions voids the applicability of analytical methods and alternative analysis is required. Improving RM&A analysis creates an avenue for engineering analysts to predict system performance and explain system behavior. Expanding the knowledge for handling reliability issues increases the reference material in the academic arena for supporting system-level analysis. New approaches to RM&A analysis support practicing engineers in evaluating real-world scenarios.

The industrial community continues to search for a simple method of handling RM&A analysis. The practicing engineer is skilled in the area of reliability and statistics,

but not necessarily the art of simulation and code writing. Advancements in the field of reliability benefit industry only if they are easily understood and applicable to their company. The importance of expanding system-level assessment is magnified when the solution is accessible to the engineer. Any changes to current reliability software must remain generic in nature, providing the practitioner an easy-to-use tool for predicting system behavior.

The Air Force continues to support ongoing research in the area of reliability simulation. New versions of simulation tools are developed and new ideas for enhancing the software are included to increase modeling flexibility. Any improvements in reliability analysis and simulation structure benefit the Air Force in testing weapons systems and support equipment. Changes to modeling tools have already been affected and continue to play a critical role in evaluating equipment acquisitions.

All areas of reliability engineering benefit from exposure and continued research relating academia to industry, and to the military sector. The cross section of analysis extending academic solutions to real-world situations can not be underestimated. The bridge that links theoretical developments to applied situations needs to be reinforced and supported with ongoing research. This research provides a chance to expand basic understanding of reliability principles and demonstrate the use of new concepts in the setting of the industrial community. The medium for interaction between these institutions is strengthened and both sectors gain from working together.

Organization

This research is presented in five chapters, including this introductory chapter which outlines the background for the problem and research goals. The first chapter also provides insight into the motivation for the research along with the importance of the results to the academic and industrial community. The following four chapters present the unique contribution of this research to the reader in an organized fashion. Each chapter has a specific function and provides the foundation for the chapters that follow. As mentioned earlier, Chapter 1 is an introduction to this study and provides motivational insight outlining the work in reliability analysis.

Chapter 2 lays the foundation for this research and provides a detailed literature review of related material. The material is organized to support the need for advancements in RM&A analysis and directs the remaining research accordingly. Background is given on reliability in general, analytical solutions and their limitations, Monte Carlo simulation, and current reliability simulation tools. Chapter 2 addresses the shortcomings in current simulation programs and indicates areas for potential research. Advancements for overcoming the current shortcomings is the grounds for the original contributions presented during this study. Finally, Chapter 2 outlines the understanding gained by new analysis and how this study is beneficial to the reliability community.

Chapter 3 is dedicated to explaining the enhancements to the RAPTOR (Rapid Availability Prototyping for Testing Operational Readiness) software. Each additional feature is described and coding changes are identified which alters the modeling behavior. The new algorithms that change the structure of the simulation code are

presented along with simple examples highlighting the impact of the added features. Although the examples provided in Chapter 3 are simple, the results of the case studies contrast the old and new software and introduce the advanced modeling concepts. The simulation code is also presented to document programming and code changes to the software and allow for replication of research results.

Chapter 4 is broken into three sections, each dealing with an original contribution to the field of reliability analysis. The first section deals with sensitivity analysis and approaching system-level analysis with limited or no available data. The case study is a submitted article based on supplementing the component's Weibull failure density with a triangle distribution. The triangle distribution parameters are much easier to estimate than the Weibull, and both cases are simulated to compare the output metrics. The sensitivity of the availability and reliability performance metrics are tracked and presented to the reader.

The second section of Chapter 4 presents a real-world case study demonstrating the capability of using generic modeling to produce timely solutions for reliability questions. The section outlines the method for comparing sparing strategies in a communications network. The practitioner is exposed to the relative ease of using a robust simulation environment to obtain accurate estimates for system behavior. Furthermore, the outcome from the simulation is incorporated into a cost equation to provide a decision-making criteria for competing maintenance strategies. The analysis demonstrates the utility of expanding RM&A analysis using a robust simulation environment.

The final section of Chapter 4 clarifies current research which discusses the time between failure and time to failure distributions. These distributions are often confused, and the resulting use of an exponential distribution causes erroneous results in many reliability applications. The outcome of incorrectly applying an exponential distribution in place of the true time to failure distribution is demonstrated by an accompanying case study. The study focuses on sparing allocation and the expenditure of logistic resources. Dramatic changes are highlighted based on the correct implementation of maintainability analysis. The study also demonstrates the importance of modeling a system correctly and the benefits of a user-friendly software tool.

Chapter 5 concludes the research by outlining the original contributions and providing guidance for future research. Additional changes to the simulation software are recommended and ideas for contributing to academia are presented. Chapter 5 suggests expanding many of the simplified case studies presented in Chapter 3. The simple scenarios can be modified to represent real-world situations and provide solutions to questions on system-level RM&A behavior. A conclusion section provides overall comments and a summary of the research. Final comments remind the reader of potential research applications and recap the contributions of the study.

CHAPTER 2

LITERATURE REVIEW

Introduction

Reliability, maintainability, and availability (RM&A) analysis are fundamental to the field of reliability engineering. Advancing the field of study continues to be an important area of research and provides the analyst with new and improved methods for handling system evaluations. This study uses the tool of reliability simulation to present new work in the area of reliability. However, the background and support for reliability includes a great deal of information leading up to the idea of simulation.

Villemeur (1992) gives a brief history of the field of reliability, starting in the 1930s with mechanical systems. Electrical systems improved reliability by simple adding redundant wiring. The 1940s brought World War II and the military focused on weapons system reliability. General Motors continued the pursuit of reliability in the industrial setting. Villemeur (1992) dates reliability as a branch of engineering to the 1950s and credits this to the field of electronics. The 1960s brought new and widely applied techniques in reliability as the field became dominant in product and process improvements. Kales (1998) credits competition with bringing reliability into focus in the 1980s and 90s. From here, reliability became a part of the total quality initiatives of companies trying to gain a competitive edge.

Many of the early studies can from examining warranty periods, etc, etc

The primary method for handling early evaluation is to use analytical methods and derive applicable equations to calculate parameters of interest. The first section of

this literature review examines analytical methods. The shortcomings are discussed along with an example of how analytical methods become cumbersome. The background on analytical methods is given to demonstrate the need for alternative solutions. In many cases, analytical methods fall short in their ability to solve reliability problems due to the complexity of the real-world problem and the size of the system in question.

The idea of using simulation in place of analytical methods is presented beginning with the basic concept of Monte Carlo simulation. From here, background is provided with respect to the limitations of using simulation as a tool for performing reliability analysis. The limitations can be overcome due to advances in simulation software, programming capability, and the power of computing technology. Individual responses to the limitations of reliability simulation demonstrate the utility of modeling and simulation to overcome limits to analytical solutions.

Once the support for using simulation is documented, additional information provides guidelines for developing a simulation environment. Current simulation languages are discussed along with methods and techniques building a generic simulation tool. This section of the literature review also addresses the user's concern when developing a robust simulation environment. These criteria are guidelines in later work when coding changes affect simulation tools and enhance the modeling capability to simulate system-level RM&A statistics. The background directs the research into the area of improving RM&A analysis through the use of a robust simulation environment.

A review of existing reliability software is included to give the reader background information on the type of modeling tools available. The review is not an exhaustive

comparison of all software packages but introduces a number of modeling efforts presented in the literature. Many of these modeling tools were developed in past academic research and impacted the level of RM&A understanding at the time of their introduction. Other packages are privately developed tools used in government research and distributed commercially through contracted sources.

The final section of the literature review summarizes reliability simulation and proposes areas for improvement being pursued in this research. The goal is to enhance current simulation capabilities, and hence, improve the level of understanding in the field of reliability engineering. The RAPTOR (Rapid Availability Prototyping for Testing Operational Readiness) simulation software is the base for modification enhancing current modeling features and capabilities. The changes to the coding structure allow additional analysis to be completed. Conditions in the real world demand new alternatives for completing RM&A analysis. The shortcomings in current capabilities open the door for the original contribution of this study and are presented as the fundamental basis for improving reliability simulation.

Analytical Solutions

Analytical solutions provide exact results for system performance metrics. The results incorporate Operations Research methods of queuing logic and Markov chains to calculate reliability, maintainability, and availability (RM&A). Josselyn, et al (1986) and Hassett, et al (1995) use Markov models to predict RM&A performance metrics. These examples provide solutions for simple, well-behaved systems. The systems must be reduced into subsystems that can be described by separate series and parallel subsystems.

Basic repair policies must be assumed which limit the flexibility of the analytical results. Also, assumptions must be made on the distribution types for the component failure rates.

Extensive research by previous authors gives analytic results for many simple systems. Determining reliability characteristics of series systems has been studied by Kapur and Lamberson (1977) and Billington and Allan (1992). Further manipulation of series systems also allows for determination of maintainability and availability performance metrics (Elsayed (1996), and Kececioglu (1991)). As seen in the reliability block diagram (RBD) below, the series system requires all components in the RBD to be in line with one another. All components must be operating for the system to be in an up, or operating condition.

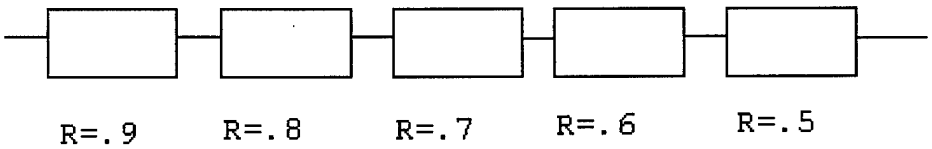


FIGURE 2-1. Series System

The reliability of this system is found by multiplying the individual reliability of each component. In the example above, the system reliability equals 15.12% (.9*.8*.7*.6*.5) and any component failure causes the system to fail.

Parallel systems have also been studied extensively, including analytical results for RM&A performance metrics (Kapur and Lamberson (1977)). Methods exist for making modifications to parallel systems such as warm and cold stand-by systems.

Stand-by systems allow for back-up units to be included in the system level block diagram. Upon failure of a component, the stand-by unit is activated and replaces the failed component during system operation. Warm stand-by units are already running and are switched into operation. Cold stand-by units require start-up and switching to be placed in the system for a failed unit. The RBD below shows a simple parallel system; one out of five paths are required to maintain operation.

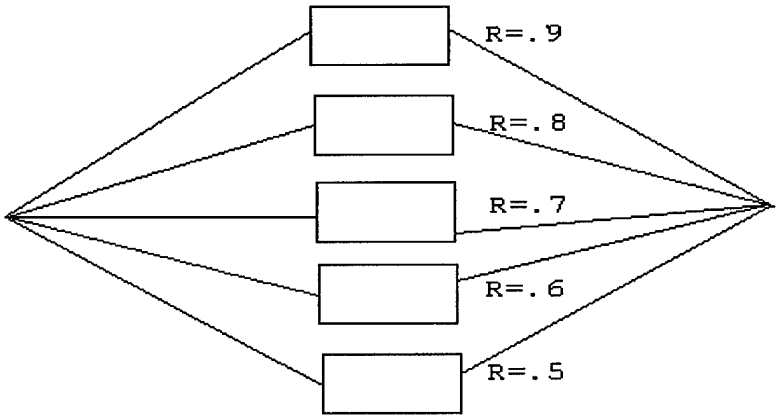


FIGURE 2-2. Parallel System

The system reliability equals 99.88%, found by multiplying (1-.9)(1-.8)(1-.7)(1-.6)(1-.5). The system does not include any stand-by components, and only one path is required to maintain system operation. Analytical solutions exist to calculate the RM&A performance metrics other than reliability, but are limited to simplistic systems with a minimal number of parallel components.

Complex systems, those including both series and parallel components, rely on alternative methods for solving RM&A performance metrics. Decomposition of the RBD

structure reduces the complex system into individual series and parallel subsystems and allows for calculation of reliability. Minimal cut paths and minimal spanning trees also give analytical results to system level reliability (Billington and Allan (1992) and Zacks (1992)). The extent at which analytical results exist is limited by the simplifying assumptions required to solve for the RM&A performance criteria.

Limitations to Analytical Solutions

Analytical methods directly solve many reliability systems and provide exact solutions. As the complexity and characteristics of the system increase, analytical solutions become cumbersome and impractical. The analytical methods can be restrictive in their assumptions and the analytical equations become difficult to solve. The following four cases are examples in which analytical solutions are deficient in solving reliability networks. Simple examples demonstrate these limitations, and simulation models are implemented to provide estimates for system behavior. Real systems are mentioned to highlight the severity of the problem as the size of the system increases. After the examples, additional restrictions of analytical models and current simulation tools are discussed.

Example 1

The first example uses a relatively simple reliability block diagram (RBD) to demonstrate the difficulty in reducing reliability networks into series/parallel systems. A five component system is constructed using a combined series/parallel structure. The following RBD can not be reduced in terms of combining component behavior due to

additional links between blocks. No two or more blocks can be combined into an aggregate component.

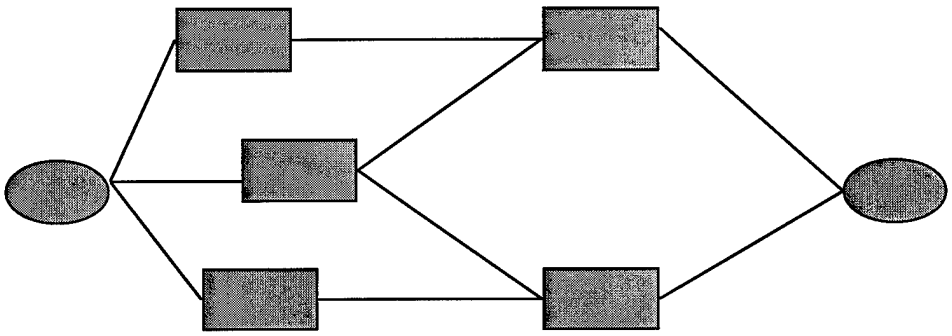
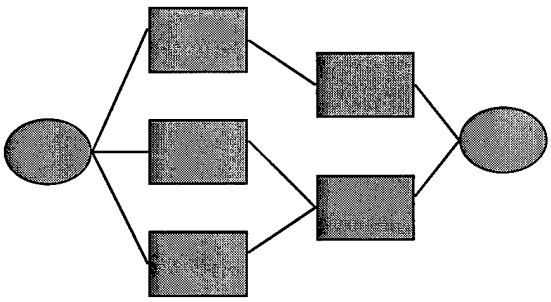


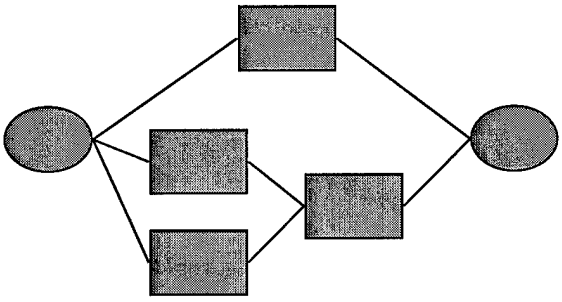
FIGURE 2-3. Example 1, Reliability Block Diagram

A direct calculation for system reliability can not be determined. The lower and upper bounds are calculated from approximating the RBD structure in a form solvable by analytical equations.

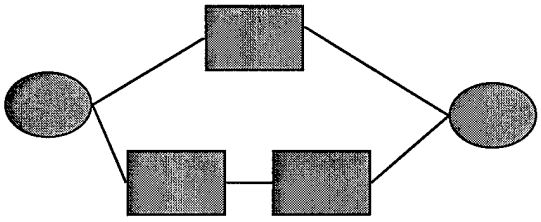
Assuming the reliability of all components is 80%, a lower bound for the system reliability is calculated using the steps shown below.



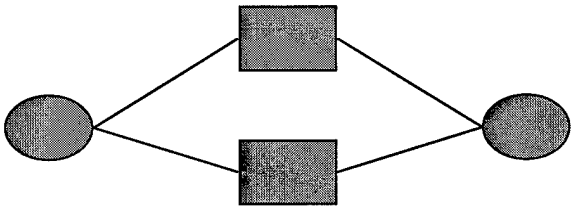
the original lower bound



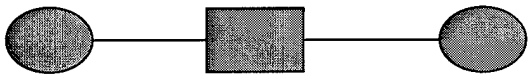
top two series nodes are combined



two lower parallel blks are combined



two lower series nodes are combined



represents the system lower bound

FIGURE 2-4. Reducing the Lower Bound of the Reliability Block Diagram

At each step in the reductions, the "new" component reliability is calculated. Eventually, the entire RBD is reduced to one block representing the original system lower bound. The final reliability value is equal to 91.6%.

Similarly, an upper bound for the system reliability is calculated using the same approach described above. The upper bound reliability is 97.0% and is derived using the RBD shown below.

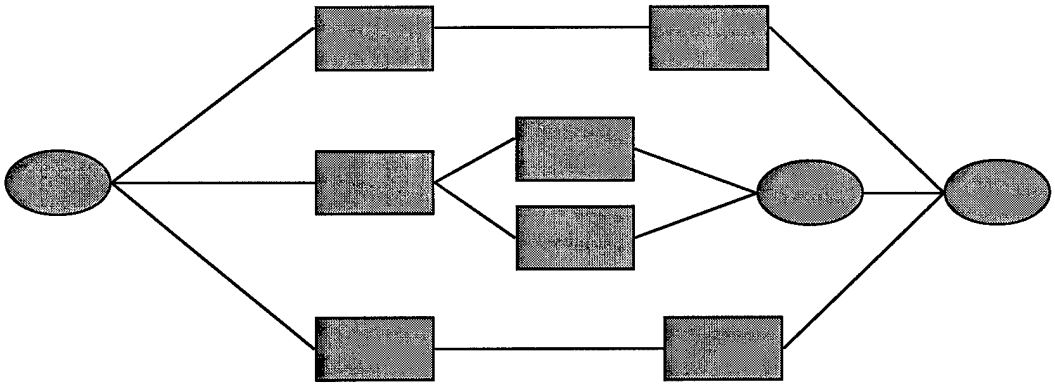


FIGURE 2-5. Upper Bound of the Reliability Block Diagram

The true system reliability therefore falls between 91.6% and 97.0%. Simulation offers an alternate approach and provides a method for calculating the original system reliability directly.

The United States Air Force evaluates new systems being purchased during the acquisition process. The maintainability aspects are tested to determine if they meet the needs of the operational unit. The Air Force Test and Evaluation Center (AFOTEC) is responsible for this analysis and using simulation on a regular basis to evaluate reliability performance. RAPTOR (Rapid Availability Prototyping for Testing Operational Readiness) is used to approximate the original system reliability. The lower bound is also estimated to support the accuracy of the simulation results. The simulated lower bound for the original system is equal to 91.2%. This represents an error of only 0.4%.

The original system is simulated directly, and the overall reliability is equal to 94.2%. This falls within the analytically bounds for the system and provides the user with an estimate for system performance. The simulation does not reduce the reliability

block diagram. Instead, the behavior of each component is simulated, and their failures and repairs are propagated through the system. The software package monitors the overall availability of the system as time progresses. The availability converges to the system reliability in the contents of the sample problem.

Real examples increase the complexity of this problem as the number of nodes, links and blocks continues to grow. The logistics analysis office in Albuquerque, New Mexico is working in conjunction with the Melbourne, Australia traffic department. The combined reliability network for that system includes over 71 nodes, over 300 blocks and over 500 connecting links. Analytical solutions are infeasible and impractical for this problem. The RAPTOR simulation software is being used to generate system performance metrics. The data itself is contractor sensitive and can not be included in this dissertation.

Example 2

The next example involves a small system composed of five components in parallel. All components are assumed independent and identical. They fail and are repaired following an exponential distribution. There are no resource limitations for repairing the system when components fail. The system is modeled using a Markovian birth/death process. Gross and Harris (1985) demonstrate theoretical results to simple systems. The reliability block diagram and the state diagram are shown below.

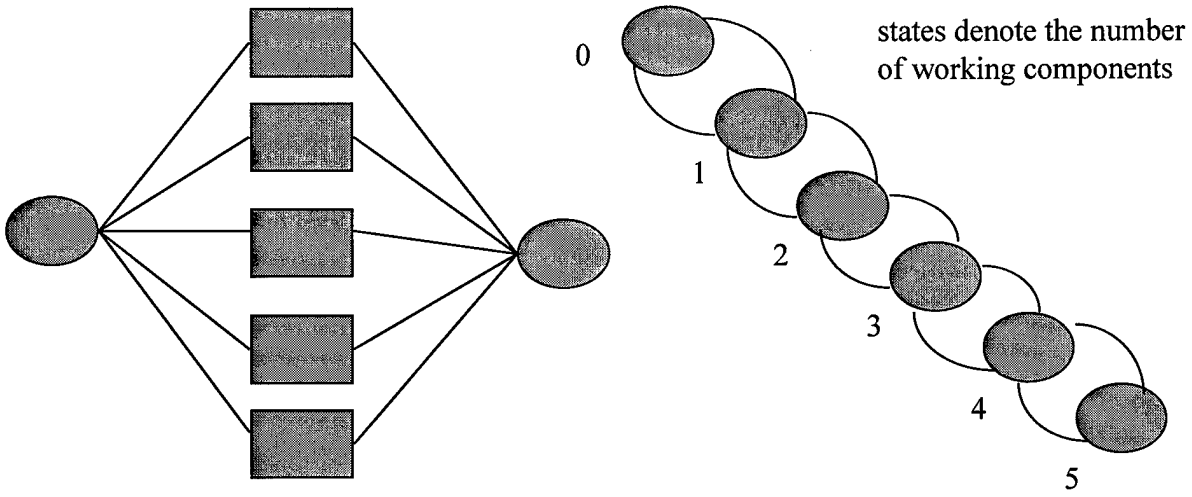


FIGURE 2-6. Example 2, Reliability Block Diagram and State Diagram

The transition matrix is derived from the state diagram using the Markovian approach. The failure and repair rates determine the transition rates from state i to j. Using the assumptions of the problem, this system is limited to six states. If the components were not identical with respect to the repair or failure distribution, the number of states would increase. The total number of states possible is equal to 2^n (n = to the number of components), or 32 possible states in this simplified example. The transition matrix for this example, with failure rate equal to 0.1 and repair rate equal to 0.05, is shown below.

State i,j		0	1	2	3	4	5		
		0	-.25	.25	0	0	0	0	.25a + .1b = 0
		1	.1	-.3	.2	0	0	0	25a - .3b + .2c = 0
Q	=	2	0	.2	-.35	.15	0	0	2b - .35c + .3d = 0
		3	0	0	.3	-.4	.1	0	15c - .4d + .4e = 0
		4	0	0	0	.4	-.45	.05	1d - .45e + .5f = 0
		5	0	0	0	0	.5	-.5	05e - .5f = 0
									a + b + c + d + e + f = 1

FIGURE 2-7. Transition Matrix and Steady State Equations

The transition matrix is used to derive the steady state equations (also shown above with a, b, … f representing states 0, 1, ... 5 respectively). These equations are then solved to calculate the long-term, steady state behavior. Assuming a one out of five system (k out of n) for the parallel redundancy, the long-term, steady state behavior is one minus the probability of being in state 0. State 0 represents zero working components, and thus, a failed system. For our example, the reliability is equal to 86.8%.

The state diagram and the transition matrix exponentially grow as the system increases inside. As mentioned before, removing the assumption of independent and identical components, the new state diagram would have 32 states. Finding the steady state conditions requires solving a system of 32 equations for a relatively simple reliability block diagram. This analytical method is further restricted by the assumption of exponentiality. The Markovian approach fails when new distributions are designated for the component failure and repair behavior.

Simulation languages are capable of generating a large variety of distribution types, such as Weibull, Lognormal and Triangle. These distributions are used to model the appropriate component behavior. The simplified example above is modeled using RAPTOR to demonstrate the accuracy of the results. The network reliability is 86.2% using simulation, within 0.6% of the true analytical value. Referring to the Australian traffic problem, the Markovian state diagram and transition matrix would be near impossible to construct. The steady state equations would generate a system consisting of 2^{300} equations. Not even today's computers are capable of solving this system in a short period of time.

Example 3

The system used in example 2 is now modified to eliminate the repair capability. The long-term, steady state reliability is therefore equal to zero. Instead, the transient behavior is the performance metric of interest. The steady state equations must be abandoned and differential equations are used in their place. Differential equations depict the system with respect to time. Again, the same assumptions are made with respect to independent and identically distributed components (failures and repairs are exponential). Notice in the steady state diagram below, there are no transitions from state i to state i + 1. The missing transition represents the absence of repair capability.

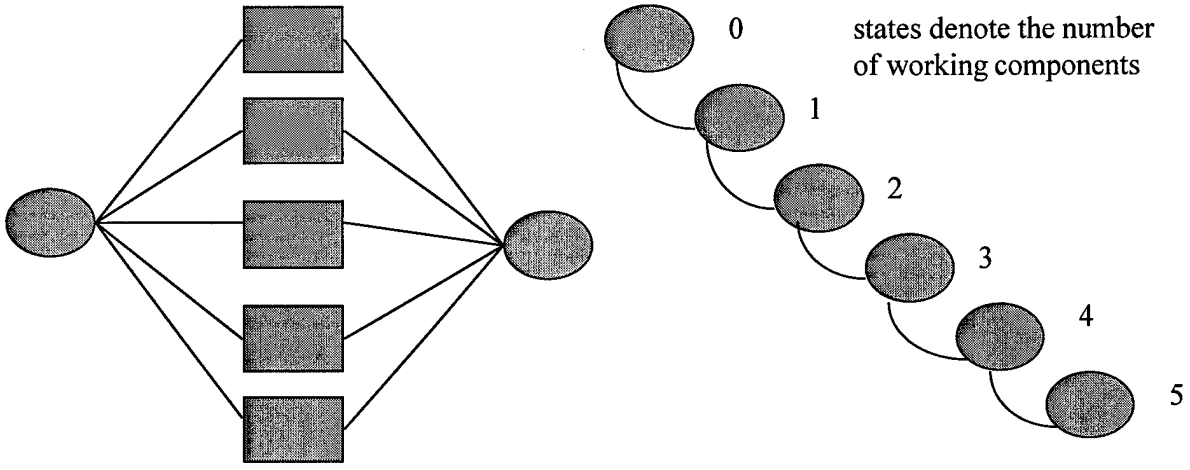


FIGURE 2-8. Example 3, Reliability Block Diagram and State Diagram

The transition matrix is generated as before, and the differential equations are constructed using the matrix entries.

State i,j	0	1	2	3	4	5	
0	-.25	.25	0	0	0	0	
1	.1	-.3	.2	0	0	0	
Q = 2	0	.2	-.35	.15	0	0	
3	0	0	.3	-.4	.1	0	
4	0	0	0	.4	-.45	.05	
5	0	0	0	0	.5	-.5	

$$p'_a = 2p_b$$

$$p'_b = -2p_b + 4p_c$$

$$p'_c = -4p_c + 6p_d$$

$$p'_d = -6p_d + 8p_e$$

$$p'_e = -8p_e + 10p_f$$

$$p'_f = -10p_f$$

FIGURE 2-9. Transition Matrix and Differential Equations

The differential equations are solved using continuous simulation in SLAM and explained by the Pristker text (1986). Appendix 2A contains the SLAM code listing. The steady state reliability does decay to zero as expected. The output of the simulation is imported into a Microsoft Excel spreadsheet, and the transient behavior of the system reliability is plotted. All components are initially set to an operating condition; the system is in state 5. The system reliability is therefore equal to one and decays to zero as shown below.

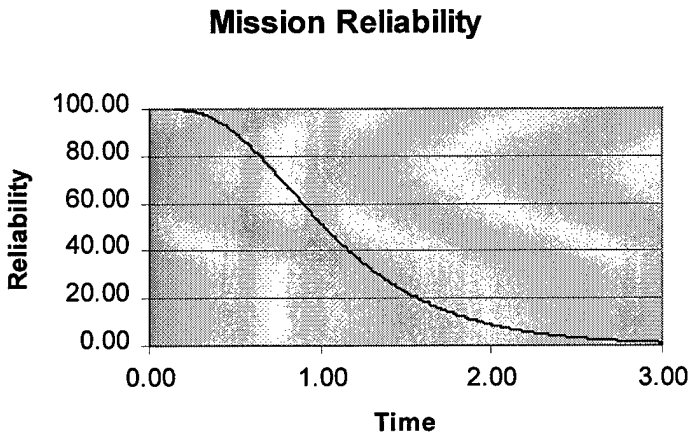


FIGURE 2-10. System Reliability Over Time

Many questions can be answered by using the provided graph. The system reliability reaches an approximate zero value within three hours of operation. The mission reliability (reliability for a specific period of time) is also read directly from this graph. The one hour mission reliability for this system is 51.7% (actual values are taken from the solutions to the differential equations). The system reaches 90% reliability at

0.5 hours of operation. These solutions provide valuable insight into system performance.

The state diagram and set of differential equations increase dramatically as assumptions are removed and the size of the reliability block diagram grows. RAPTOR is capable of modeling this system and provides the same type of output results. Instead of simulating for a converging steady state condition, a pre-defined mission length is specified by the user. The simulated one-hour mission reliability is 53.8% and the 0.5-hour mission reliability is 91.0%. The entire transient behavior could be plotted by incrementing the mission length and graphing the system performance.

The same logistics analysis office is using mission reliability to predict performance of the Air Force B-2 bomber aircraft. The performance metrics exact values are not included due to their sensitive nature. The RBD for the entire system includes over 50 nodes and over 200 blocks. The transition matrix, if obtainable, would generate a system of 2^{200} differential equations. However, the mission reliability of the B-2 is simulated using RAPTOR, and the results are considered an integral portion of the aircraft evaluation. Additionally, RAPTOR provides reliability estimates without accumulating flight hours on the aircraft. The financial cost of flying each mission is extremely high. The simulated results provide answers for planning and logistics until real sortie data is generated.

Example 4

A final example is used to show the limitations of current simulation software. Although RAPTOR is a very useful and capable program, there are a number of

limitations and assumptions driving the model behavior. Specifically, RAPTOR does not currently handle limited resources or assigned priorities. A five component, parallel system is used for this example. The analytical solutions are calculated to verify the accuracy of the SLAM program written to model network reliability. The SLAM listing, control file and output are in the Appendix 2B.

All components are independent and identically distributed. The failure rate is two hours and repair rate is at one hour, both distributed exponentially. The repair resources are limited to one server, causing the repair rate to be constant in the transition matrix. The same Markovian approach is used as explained in example 2. The long-term, steady state reliability of this system is equal to 39.4%. The SLAM simulation reliability is 38.7%, within 0.7% of the true system performance.

The analytical approach becomes cumbersome and difficult to implement as the number of components increases. Relaxation of assumptions also hinders analytical methods as previously discussed. The SLAM simulation program is capable of handling a wide variety of distribution types and components with different failure and repair distributions. As before, the simple case is presented to gain support for the accuracy of simulation. System complexity and size surpass the capabilities of analytical methods. Software, such as the SLAM program, provides a modeling structure for generating reasonable performance metrics.

The SLAM program provides additional flexibility by allowing priorities to be placed on the maintenance queue. With only one repair resource, a priority rule is implemented to determine the order of repair operations. Changing the repair rate for the

five parallel components modifies the example. Analytical methods are capable of solving this system, but are ignored. The purpose is to demonstrate increased capability of the SLAM program, above the current capability of reliability simulation tools, such as RAPTOR. Three blocks have mean repair times of one hour, while the other two have mean repair times of five hours.

The reliability of this system, with no priorities, is equal to 16.9%. This is a considerable drop from the original 39.4% cited above and is much greater than would be expected by changing two component repair rates. The first in, first out (FIFO) queuing strategy does not optimize the use of the single repair resource. FIFO is replaced by a priority strategy based on the lowest repair time. This gives three of the components a higher priority and uses the limited repair server more efficiently. The system returns to an operational condition more quickly with this priority. The new reliability is 32.9%, which is much closer to what is expected, given the minimal changes to the details of the system.

Additional Limitations of Analytical Methods

These simple examples highlight many of the limitations of analytically methods. Not all reliability networks can be reduced to simple series/parallel structures. In many cases, the analytical methods are limited strictly due to the size of the problem. The exponential growth of the state space decreases the likelihood of finding solutions to steady state or differential systems of equations. Simulation software packages are also limited in their flexibility and capabilities to model realistic systems. Additional factors can further limit the applicability of the analytical methods and current simulation tools.

Logistic considerations are very important to the maintainability aspect of a reliability network. Inventory behavior, spares information and other administrative and logistic considerations surpass the simple assumptions required for analytical solutions for repairable systems. Gonzales-Vega, et al (1988) demonstrate the potential for using simulation to model logistic effects on complex systems. The concept of spare replenishment is also an issue. Aircraft supplies were FEDEX'ed into Saudi Arabia during Desert Storm on a daily basis. The impact on sortie generation and aircraft availability was tremendous.

Component dependency is not mentioned in any of the examples. This creates an entire new area of problems for both analytical solutions and simulation tools. Dependencies among component failure and repair behavior are very realistic. Maintenance concepts often recommend repair of related components at the time of failure. Component failures may cause increased failure rates of dependent components or cause failure immediately. The dependencies may be stochastic in nature themselves, implying that not all failures of a component cause dependent failures, only a proportion of them.

Another area of difficulty surrounds the increased or decreased reliability of a system due to reliability growth, learning curve, personnel turnover or design fixes. The system reliability may increase with one repair, but decrease in the very next step. These are examples of "repair better than new" and "repair bad as old" maintenance concepts. This dramatically increases the level of complexity as each component may behave differently. Maintenance strategies, such as remove and replace versus repair, also effect

approaches to solving system reliability. Current simulation tools model one strategy or the other, but do not combine both into the same software.

There are many limitations to using analytical methods to solve reliability networks in real-world applications. The examples given demonstrate a number of these with relatively simple reliability block diagrams and give insight into the increased difficulties as the size and complexity of the RBD increase. Exponential growth of the state space, priorities of limited resources, logistics considerations and component dependencies are all issues to be considered. If simplifying assumptions are feasible; however, many systems can be solved directly using analytical models.

A simulation approach is proposed as an alternative to analytical models when reliability networks violate the required assumptions. The results of simulation are an approximation of system performance (due to their stochastic nature), but are shown to reasonable accurate in the given examples. Law and Kelton (1982) address simulation and methods for increasing the accuracy of the results. Although analytical methods provide exact solutions, often times, a reasonable approximation is better than no solution at all. Simulation tools are not error free. This approach suffers from many of the same limitations mentioned above. There is a continued need to improve the modeling capability of current simulation tools to account for realistic behavior of real-world systems. With increased capability, understanding of reliability networks can continue to improve.

Monte Carlo Simulation

In order to overcome limitations in analytical solutions, analysts take advantage of reliability simulation. Foster, et al, (1981) believe "simulation analysis is a natural and logical extension to the analytic and special mathematical models". The simulation tools are based on Monte Carlo simulation. The technique involves consecutive draws from varying distributions representing the failure and repair rates. From these distributions, the simulation predicts the actual system performance. Monte Carlo simulation offers a possible solution for determining reliability, maintainability, and availability solutions to complex systems when analytical methods fall short (Kovalenko, et al (1997) and Leitch (1995)).

Monte Carlo simulation applies to all types of systems, from simple series block diagrams to complex systems with series, parallel, and stand-by units integrated into a reliability block diagram (RBD). Hwang, et al (1981) review reliability modeling for complex systems and cite Monte Carlo simulation as a credible alternative to analytical methods. Many examples in the literature demonstrate successful solutions using simulation. Conway and Goyal (1987), Foster, et al (1986), and Windebank (1983) use reliability simulation to attain system performance metrics.

Comparing analytical and simulation results from simple systems provides an appreciation for the accuracy of Monte Carlo simulation. In example 2 in the previous section, the true (analytical) reliability is 86.8% and the simulated reliability is 86.2%. Accurate model design, sufficient run length, and replication reduce variation and offer better approximation to exact analytical solutions (Banks, et al. (1996)). The complexity

of the system and the changing characteristics effect the overall performance of Monte Carlo simulation models. However, simulation offers a chance to approximate RM&A performance metrics when analytical solutions become infeasible as demonstrated by the previous examples. The RAPTOR simulation tool depends on using reliability block diagrams (RBDs).

Basically, the RBD simulation approach uses an event calendar and random distribution draws to represent failure and repair times for individual components. A reliability block diagram is a graphical representation depicting the structure and connection of the blocks, nodes and links of the entire system. The components are manipulated and their effects are propagated through the entire system. The simulation monitors the impact on the overall system reliability and availability performance. The simulation language must be capable of placing each component in an operating or failed state. The simulation continues until the next event causes a possible change to the system state. Additional features are added to the simulation to more accurately depict the real-world system and provide increased confidence in system performance estimation.

There are shortcomings to modeling using Monte Carlo simulation and reliability tools. Ascher and Feingold (1984) list eighteen shortcomings of modeling due to system complexity. The limitations of reliability simulation are discussed in general and present an obstacle to accurate system modeling. Many of the limitations can be handled by appropriate modeling and coding algorithms. The limitations from Ascher and Feingold, along with a brief response to each critique, provides insight into eliminating the

shortfalls by improving reliability simulation. A number of the shortcomings have been overcome and solutions have been implemented into current simulation tools such as RAPTOR.

Shortcoming to Monte Carlo Simulation

Fifteen of the eighteen shortcomings from Ascher and Feingold are discussed. Each critique is briefly summarized along with the accompanying justification for eliminating the shortfall. The following arguments refer to simulation in general and not to a specific simulation tool or software package. The critique responses are phrased with respect to simulation in general, and use the applicable terminology. The goal is to explain how each shortcoming could in fact be modeled using simulation practice and Monte Carlo simulation.

Critique 1 - All parts are not in series.

Response 1 - The structure of the systems is defined according to the reliability block diagram. Series and parallel components can be constructed and affect the system differently. A series component failure takes down the entire path, while the parallel components are dependent on the redundancy in the system. The simulation simply propagates the individual failures throughout the system to determine the effect on RM&A parameters.

Critique 2 - There may be incomplete repairs where the real problem is not corrected during the first repair attempt.

Response 2 - The original repair is made according to the user-defined repair distribution. The system performance is affected by the appropriate delay value for the

component repair. After the repair is complete, the success of this action must be tested. A random uniform distribution draw is compared to a pre-defined success probability to identify this repair action as a success or failure. A successful repair is considered 100% complete, and the next time to failure for this component is evaluated based on the appropriate distribution and mean time to failure. A failed repair, as indicated by the criticism above, results in a substandard repair action. The mean time to failure could be adversely affected in two ways. The time to next failure is immediate implying the repair action was a complete failure. Or, the time to next failure of this component is modified to occur sooner than expected due to the incomplete repair. Both of these situations are handled by defining an incomplete repair factor (between zero and one). Multiplying the mean time to next failure by this factor shortens the component life after the incomplete repair. This situation is considered a "bad as old" repair capability for the RBD simulation.

Critique 3 - There may be damage/failures caused by improper system operation, scheduled maintenance or repair. In contrast, there may be improvement due to particularly effective repairs. Moreover, there may be a strong effect due to improved operating and service manuals as well as learning curves for operators and maintenance men. In addition, there may be significant "forgetting" due to high personnel turnover.

Response 3 - There are multiple ways to address this problem. First, assume that the failure distribution includes all types of failures and represents the behavior of the true system. This assumption is somewhat unrealistic and alternative approaches will be discussed. Second, the user constructs the RBD in such a fashion using "n" parallel

components to define each failure mode (failures due to improper system operation, etc.). All "n" of "n" components must be working in order for the parallel structure to be successful. After each failure, the other parallel components failure times are reset and new mean time to failures are drawn for each distribution to represent the renewal of the component. Otherwise, after each failure the other parallel components failure times are NOT reset. This represents continued possibility for immediate failures of this component due to other failure modes. Neither of the first two approaches handle the situation of learning curves or "forgetting" due to turnover. The third approach uses simulation to manipulate the mean time to failure in a positive or negative direction depending on the circumstance. This is considered a "better than new" or "bad as old" repair policy. The learning curves increase reliability or delay the next time to failure to prolong the life of the component. The time to next failure is determined by multiplying the appropriate distribution draw by a reliability or learning curve factor (greater than one). The same procedure is used to simulate "bad as old" or decreased repair due to "forgetting"; the mean time to failure distribution draw is multiplied by a degradation factor (less than one). Either way, the component would have to be defined, specifying to which influence, improvement or degradation, the failure parameter is susceptible.

Critique 4 - In the case of redundant systems, nonzero repair times must be considered. Moreover, logistics time awaiting spare parts may greatly exceed active repair times.

Response 4 - Simulation handles this situation with relative ease. Before a failed component in a redundant system is specified as operational, the simulation delays by an

appropriate value to represent the repair time. Logistics time is handled by tracking the availability of spare parts. Before repairing the failed component, the model requests a spare part. If one is available, the repair begins after the proper delay due to transportation time. Otherwise, the repair is halted until a new spare can be ordered, received and delivered to the system. All these additional delays are handled during simulation by increasing or adding the delay times to the total repair time for the component. This models the appropriate logistic behavior of the defined system.

Critique 5 - System failures may damage or cause failures of other parts. This will lead to either simultaneous part replacements or multiple system downtimes.

Response 5 - The simulation incorporates a concept of "stream dependencies", allowing the user to define which components are related. Flagged components are dependent upon each other due to failure or repair circumstances. At the failure time of an individual component, the simulation determines if the stream components are effected. A uniform distribution "test" draw is compared to a pre-defined success limit. If the failure impacts the "stream" the simulation simultaneously fails all of the related components by setting their time to next failure equal to the current time and placing them in a failed condition. Alternatively, the failure rate for the dependent components could be increased, shortening their expected time to failure.

Critique 6 - Repairs may be made by adjustment, i.e., no parts are replaced.

Response 6 - The simulation could take into account two maintenance concepts when components fail: one in which spares are required and the components are replaced during maintenance, and a second in which no spare parts are required and the

component is simply repaired by adjustment. The user specifies the proportion of the time in which replacement maintenance, using spare parts, is required. One minus this proportion represents maintenance made by repair or adjustment of the failed component. The failure is followed by a uniform draw to determine the replace or repair action required. Depending on which action is required, the component experiences a down time according to the repair or replacement distribution, both specified by the user. The user could specify two completely different distributions for the mean time to repair and the mean time to replace maintenance concepts.

Critique 7 - Stresses like on/off cycling and environmental effects may be comparable to, or more important than, system operating times. Moreover, these stresses may change over time; for example, there may be seasonal effects.

Response 7 - This situation is handled by placing a series component at the beginning of the RBD, which represents the failure rate due to cycling and environmental effects. Changing this parameter over time is done using the "better than new" or "bad as old" behavior described previously. More importantly, the series component causes the entire system to fail due to its placement at the beginning of the system; this is expected due to the nature of the failure. An alternative approach uses simulation to model the system while simultaneously modeling cycling. This is done by placing a counter on the series component and simulating a failure after the counter reaches a pre-defined number of cycles (one cycle representing one run or "x" number of operating hours). Another option includes an event node representing the beginning series node. A random draw is compared to the pre-defined probability of success each time the system begins

operation. Therefore, this node only "fires" once at the beginning of the simulation, but fails at different points in time versus failing after a specified number of cycles while using a counter. Finally, the seasonal effect is handled using a "phased" simulation approach. This approach allows the failure rate parameter to change after a certain number of operational hours elapse. The RBD is defined over a range of time values, and the simulation makes appropriate distribution draws dependent on which "phase" the system is currently operating.

Critique 8 - Different modes of system operation may be susceptible to the same environmental stresses.

Response 8 - Different operating modes are represented using different "phases" as mentioned in the previous response. The separate "phases" each include an environmental series component to represent the failure rate of this additional stress common to all "phases". The previous condition of the environmental stress node is passed from phase to phase as the simulation continues.

Critique 9 - Two or more system parts may be susceptible to the same environmental stress.

Response 9 - This is handled in two ways: model the common failure as a series component prior to the two or more susceptible parts, or model the two or more susceptible parts using a "stream dependency" as described previously. Using a series component to represent the common environmental stress is appealing if the failure distribution can be separated in this fashion.

Critique 10 - Unusual failure modes or common cause failures may affect normally redundant channels.

Response 10 - This is directly simulated using the series component placed prior to or directly after the redundant path to represent the unusual failure mode. As mentioned before, failure of this node causes a failure of the entire path. The RBD "tricks" the simulation into believing the redundant path also includes the common cause component and requires it to be operational.

Critique 11 - Reliability improvement may occur due to installation of design fixes or the debugging of bad parts.

Response 11 - Once again the concept of "better than new" repair assists in this situation. The mean time to failure is manipulated or the shape parameter itself is modified after "x" number of failures to represent this condition. A limit is placed on the amount of improvement to avoid continued improvement. The improvement factor (pre-defined greater than one) is decreased after each repair causing the scale parameter or mean time to failure to converge to a constant value. This provides the desired behavior of reliability improvement.

Critique 12 - Replacement parts are not necessarily from the same population as the original parts.

Response 12 - Using simulation, spare parts have their own failure and repair distribution separate from that of the original components. The user specifies both distributions and their accompanying parameters when defining the system and constructing the reliability block diagram. This concept also addresses the previous

criticism of reliability improvement due to installation of design fixes. The reliability improvement results from having a spare population with higher reliability and maintainability characteristics (free from design bugs).

Critique 13 - There will be situations where all parts are within their specifications but the system isn't, or conversely, the system may be within "specs" but one or more parts is not.

Response 13 - A "system metric node" is added to the simulation model to account for this factor. This node is adjusted as the individual components approach their respective failure times or are repaired. The system metric node is a cumulative tracking node set to fail at a pre-defined limit. As more and more components approach or reach failure, the metric node increases until its failure causes the entire system to be down. Conversely, the individual components can still fail without pushing the cumulative system metric node to a point of system failure. This simulation increases in complexity due to additional updating of a cumulative system metric node. Most simulations consider the components of the system either up or down, thus affecting the system in a similar manner. Drift failures and "spec" limits are not considered during simulation.

Critique 14 - The system may be susceptible to drift failures or instabilities of one of more of its constituent parts.

Response 14 - Each component of the simulation maintains its own cumulative tracking metric. At different points in time the component metric is either increased or decreased, representing the drift behavior. Once the metric increases past the pre-defined limit, the component fails and affects the entire system accordingly. The drift is

determined by a specified time series. The drift may be random or correlated to the previous position of the component metric. Currently, most simulations consider instantaneous failures, which places components in a failed state versus the drift behavior described above.

Critique 15 - There may be degradation in shipping, storage or handling.

Response 15 - The simulation performs a setup test to evaluate which components are effected by poor shipping, storage, or handling conditions. Comparing a uniform (0,1) distribution draw to a pre-defined success probability tests each component. If the component fails, the simulation sets the mean time to failure equal to zero or reduces the mean time to failure to represent an increased initial failure rate. If the component passes, the simulation continues under normal conditions and the mean time to failure is calculated by the appropriate distribution draw unaffected by shipping, storage or handling. This procedure is completed at the beginning of each simulation as part of the setup or each time a spare part is required for repair. Spare parts are also susceptible to these same failure conditions.

Critique 16 - Overhauls do not necessarily restore a system to as good as new condition.

Response 16 - The simulation adjusts the failure rate after an overhaul has been completed. The time to next failure is shortened by an appropriate amount to signify an imperfect maintenance operation. The program monitors each repair action, and if the repair is labeled as an overhaul, the modified failure rate is used. Otherwise, a good as

new condition exists and the failure rate determining the next time to failure is unaffected.

Critique 17 - When a system is opened for repair, parts in poor but unfailed condition may be replaced. Therefore, affected sockets are not be modeled by the renewal process even if the replacement parts are selected from the original population.

Response 17 - This factor is incorporated into simulation using the "stream dependency" concept previously described. When a component fails, all dependent or related components are tested to determine their condition. The current simulation time is subtracted from the time to next failure for each component to determine its remaining life. If this value is below a pre-defined limit (indicating poor but unfailed condition), the component is immediately failed and thus placed on the repair event chain. If this value is above the pre-defined limit, no action is taken and the component continues to operate until its original time to failure. Therefore, the entire "stream" now represents all the components in a given socket or specified subsystem, which are evaluated at each dependent component failure. One factor to take into consideration is to determine whether or not repairing a poor but unfailed component causes the entire system to fail. If the dependent components are in a parallel path, one failure may not fail the entire system. However, if all the poor components in the parallel path are immediately failed to allow for socket repair, this could adversely affect the system condition. If a component causes system failure, it would not be repaired during socket or subsystem maintenance. Therefore, the simulation evaluates the dependent components before repairing those found in poor but unfailed condition.

Critique 18 - There is an overemphasis on steady-state reliability.

Response 18 - The advantage of using simulation is that stopping conditions can be changed as determined by the user. The reliability is reported based on the ending simulation time. An extremely long simulation represents steady-state reliability, where as the user can specify the ending time for a shortened life expectancy. As an example, the simulation stops after one hour and checks the up or down condition of the system. Providing statistics on this condition indicates the one-hour reliability of the system versus allowing the system to continue to accumulate hours. The one-hour simulation is replicated a large number of times to obtain an accurate estimate of reliability.

A Simulation Environment

Although simulation can model any real-world characteristic given an ample amount of time and effort, this is not the desired result. The goal of modeling is to accommodate as much detail as possible while allowing the software to run quickly and provide accurate results. RBD simulation has great potential for predicting large-scale system reliability and availability. This method easily incorporates the concepts of additional delay times (administrative and logistics functions), sparing information (inventory control) and creative maintenance policies (remove and replace versus repair maintenance and "better than new" and "bad as old" repair conditions). The increase in simulation potential aids engineering in advancing the understanding of RM&A analysis.

Repetitive coding and model building limits the use of Monte Carlo simulation. Many practitioners ignore the potential advantages simulation has to offer due to the burden of developing an appropriate simulation model. There are additional skills

required to create a modeling tool to accurately depict a given system. The need remains, but the ability to continually develop reliability models limits the use of simulation as a viable alternative. Therefore, the desire to create a simulation environment exists which would take advantage of similarities between modeling constructs. The goal is to provide a modeling framework capable of evaluating RM&A metrics with little or no coding required by the analyst.

Initial attempts to model system behavior with basic languages such as FORTRAN and Pascal prove the validity of developing a simulation environment. Deborah Burns (1994) developed a macro language simulation tool for analyzing series systems. The components follow different failure and repair rate distributions. Her research also looked at making the simulation package quick and easy to use, with little programming skill required of the user. She mentions the need to incorporate reliability simulation into the early phases of product creation and evaluation during the research and development phase. Recommendations for further research rested on the addition of parallel and complex structures to her simulation package.

Gonzales-Vegas, et al (1988) demonstrate simulation potential for analyzing complex systems using a Pascal coding structure. The primary contribution to reliability prediction and analysis using simulation is the inclusion of previously ignored logistic factors and concerns. Transportation and inspection delay times, alternative repair facilities, and varying the number of repair servicemen are all additions to her simulation methodology. These factors represent characteristics of real-world systems and should not be ignored. Any new work in the area of reliability simulation must consider logistic

constraints in order to depict the real system and calculate reliability and availability metrics.

The effort by Gonzales-Vega shows the amount of time required to develop a simulation methodology capable of performing a detailed level of analysis. At the time of her research, simulation languages were not as robust as they are today, and much of her effort was spent in developing modeling algorithms from scratch. Today, simulation languages such as SIMAN and SLAM have progressed to allow for easier manipulation of event calendars, downing events, schedules, and system statistics. Simulation languages provide a powerful alternative for code development in creating a reliability simulation tool capable of predicting system performance. Foster III, et al (1986) discuss using these language to model large scale systems.

SLAM and SIMAN represent queuing languages. The failure and repair of components of a reliability block diagram (RBD) are modeled by placing these activities on an event calendar. Prisker (1986) presents simple examples of queuing models such as bank tellers servicing customers on a daily basis. Similarly, reliability simulation takes advantage of the queuing structure to identify components waiting for repair. Also, the component's next failure is scheduled and the appropriate statistics are calculated each time the state of the system changes. The process describing the each state change is coded as a separate subroutine. The subroutines take advantage of pre-defined routines and are implemented in the proper sequence. Many difficulties arise from using process oriented simulation languages to model reliability networks and hinder the practitioner's use of software packages such as SLAM and SIMAN.

The immediate limitation to all simulation languages is the up-front learning curve. This problem is magnified when dealing with process oriented simulation languages such as GPSS, SIMAN and SLAM. The engineer wishing to perform network reliability analysis must be an expert in reliability and simulation. Writing simulation code is not as simple as one might expect. Additional problems with input requirements, output analysis, random number generation and statistical significance dramatically increase the learning curve. The naïve programmer is bound to generate errors in the system performance through coding mistakes.

Simulation languages are also restricted due to limitations from simplifying assumptions. Process oriented simulation languages have pre-defined procedures and algorithms to handle many of the interactions during reliability network analysis. Routines for delaying the system, scheduling failures and repairs, and requesting resources are all part of the advantage of using such a language. This advantage quickly becomes a disadvantage if the system does not conform to the assumed behavior. This is another reason why the learning curve is so steep for these simulation languages. The programmer must fully understand all pre-defined procedures and the necessary assumptions in order to use them. All to often, the assumptions become restrictive when the system is not small and simple.

An advanced programmer can try to correct the limitation above by writing new subroutines and procedures. Usually, this must be done from scratch since the source code is not available for modification. The degree of difficulty involved quickly makes this alternative far too cumbersome. Without user-written subroutines, modeling the real-

world characteristics of a reliability network can be difficult. Simplifying assumptions must be made to fit the proposed system into the class of problems the simulation language is capable of solving.

Finally, as with all simulation languages, the amount of simulating power and computer resources can be a limitation. Advances in computer technology struggle to alleviate this problem. As computing power increases, the level of detail and size of simulation programs increases accordingly. The user continues to raise the expected level of performance for simulation models. Models once adequate for analyzing reliability networks are now considered crude approximations with limited flexibility and questionable capability of modeling realistic network scenarios.

Generic Simulation Environment

The concept of creating a generic simulation tool gives the analyst the next step in flexibility for modeling system behavior of reliability, maintainability, and availability performance metrics. Simulation experts in the field of manufacturing rely on re-use of simulation code and depend heavily on the idea of creating a generic simulation tool or "intelligent simulation environment" (Ozdemirel and Mackulak (1993)). Shifting this concept to the industrial engineer concerned with RM&A performance of a particular system provides additional benefits to the practitioner. Actual code writing and model development no longer threaten the system analyst. The practitioner need only understand the system behavior and be capable of using a generic simulation tool.

Mackulak and Cochran (1994) identify fifteen features of using a generic simulation approach in manufacturing. Features for generic simulation tools in RM&A

parallel many of these same requirements which were defined for simulation in the manufacturing setting. No user coding, interactive capability through menus and windows, automated model execution, and automated output analysis are just a few of the parameters which relate to an effective RM&A simulation tool. Advancements in generic simulation tools lead to developing an overall robust simulation environment. In this environment, the practicing reliability engineer would be capable of obtaining simulated system performance with minimal programming skills required to write modeling code.

The primary difference between creating a robust simulation environment and simulation tool deals with the user interface and input capabilities. Basically, the generic simulation tools run the software driving the simulation environment, but the added user interface allows for easy manipulation of input requirements. In a separate study, Mackulak, Cochran, and Savory (1994) focus on the requirements for important features of a simulation environment. Friendly user interface, generous graphics, and windows and menus are the top three listed features. Once again, these same requirements transfer to the industrial engineer looking to use simulation to provide assistance in RM&A analysis of complex systems.

Understanding the user's requirements drives the ability to develop a robust simulation environment with generic implementation of model building and code execution. The reliability engineers require interaction through system descriptions using reliability block diagrams (RBDs). Individual components require multiple distributions for describing failure and repair rates. Additional features such as administrative and

logistic delay, system dependencies, and sparing information represent obstacles to simple programming and model building. The goal is to provide an RBD-driven simulation package capable of predicting system RM&A metrics for real-world systems.

Review of Existing Simulation Software

Early studies at Texas A&M University showed the potential for modeling and simulation applied to evaluating the reliability of large, complex systems. John Blackstone (1979) used GASP-IV simulation language to develop a generic model which was used in simulating electric power generating systems. Pedro Resto-Batall (1982) extended this research to include a study of the input variables and development of a simulation-stopping rule. Ofelia Gonzales-Vega (1987) created a simulation methodology for evaluating reliability of large-scale systems. Her generic model approach includes logistics considerations, such as inventory, transportation, and facilities.

Reliability simulation is not limited to academia alone; industrial applications also demand similar software for RM&A analysis. Glynn (1989) uses GSMP Formalism for defining the reliability software developed for use in discrete event simulation. Additionally, Gopalokrishnan (1985) presented a Discrete Event Reliability Maintainability Availability (DERMA) Model at the TIMS/ORSA Joint National Meeting. The industrial community supports reliability simulation to improve product and process development. Similarly, government agencies also use generic modeling in evaluating system behavior.

Government agencies have developed a number of simulation tools for reliability analysis. The National Aeronautics and Space Administration (NASA) Lewis Research Center initially developed Event Time Availability, Reliability Analysis (ETARA) software to evaluate the Space Station Freedom Electrical Power System (User's Manual (1991)). The internally developed, DOS-based software provides availability parameters such as state availability, equivalent availability, and continuous state duration. The program also simulates spare allotments and spare replenishments for a re-supply cycle. The input requirements include a reliability block diagram representation of the system and distribution specifications for failure and repair rates. ETARA limits distribution selections to Weibull or Exponential (User's Manual (1991)).

Additionally, improvements to the ETARA software led to NASA's development of Availability, Cost and Resource Allocation (ACARA) simulation program (User's Manual (1993)). The program takes into account costing considerations that were ignored in the ETARA software. ACARA's improved capabilities model life cycle cost and resource allocation. The primary goal of the simulation package is to achieve optimum system performance based on scheduling component repairs. ACARA is similarly limited by the choice for failure rate distributions. The Weibull is used to model component down time for all situations.

The Air Force Operational Test and Evaluation Center (AFOTEC) maintains ownership of the Rapid Availability Prototyping for Testing Operational Readiness (RAPTOR) software. The Studies and Analysis Division (SAL) of AFOTEC developed this software for use in operational test and evaluation of new weapons and support

equipment being purchased by the Air Force (User's Manual (1995)). RAPTOR leads the field of reliability simulation tools and sets the standard for software capability in performing RM&A analysis.

RAPTOR is a modeling framework controlled completely by a "point and click" graphical user interface and is capable of quick analysis on RM&A characteristics of the defined system. The practitioner maintains great flexibility in choosing between fifteen distributions to describe the failure and repair behavior of the individual components. The software handles sparing information, administrative and logistic delays, stand-by units, series, parallel, and any combination of reliability block diagrams (User's Manual (1995)).

The PC Windows-based simulation software executes rapidly depending on the size and complexity of the system. RAPTOR allows long-term analysis of reliability, maintainability and availability performance metrics as well as "mission-based" scenarios, with a specified time duration. The software graphically depicts the changing conditions of the system, allowing the users to visually examine the operating system during simulation (User's Manual (1995)). This enhances model building, validation and presentation capabilities above and beyond other simulation packages.

Although RAPTOR leads the field in the area of reliability simulation, a number of deficiencies limit the potential use of RAPTOR in the industrial setting. Primarily, the industrial community relies heavily on cost analysis to evaluate system performance, whereas RAPTOR omits cost analysis entirely. The test and evaluation analysis driving the RAPTOR development excludes a cost estimation requirement. Other limitations

such as individual block dependencies and phased implementation are being evaluated for inclusion in RAPTOR upgrades. AFOTEC continues to improve their RAPTOR software and expects to release upgraded versions as modifications are completed.

The "Ideal" Software Package

No software package answers all issues of criticism or provides every analyst with the exact simulation environment they desire. However, certain capabilities and features provide worthwhile benefits to the practicing reliability engineer. Many of the software packages discussed begin to fulfill the desired simulation environment. The key to improving reliability analysis is to increase the modeling capability of simulation tools. One of the primary requests from practitioners is to limit the programming skill required in using reliability simulation.

The practitioner prefers a simulation package with generic modeling capabilities. The input medium of choice remains menu driven with "point-and-click" features allowing easy changes to system construction. Reliability engineers deal with reliability block diagrams; and therefore, the software package must work in the same manner. After defining the system using an RBD approach, the simulation package should simulate operation to accurately represent failures, repairs, and logistic considerations. The coding algorithms should be hidden from the user along with the programming language itself.

The output from a generic simulation tool should be independent of the input parameters and model specification. Reliability, maintainability, and availability statistics make up the core output parameters demanded by engineers. Additional

parameters include sparing information, cost considerations, and detailed reports of the system activities. Similar to the input requirements, the practitioner prefers menu driven output in the form of graphs and tabulated information.

Simple, interface features rank high on the list of requirements for a generic simulation environment. Storing and retrieval capabilities of defined systems, input specifications, and output results demonstrate a continued push for a windows-like simulation package. As mentioned earlier, many of the existing software packages already provide these capabilities. The RAPTOR simulation tool is windows oriented and capable of executing on the PC with the user interface handled graphically as desired.

Regardless of the user interface capabilities or generic code implementation, no software package provides useful results without realistic representation of real-world systems. The simulation must accurately describe the RM&A characteristics of complex, system behavior. Realism, however, is a trade-off. No simulation model represents real-world systems 100%. The more realistic the simulation model, the more believable the simulation results. The confidence in the simulation results increases at the same rate as the robustness of the model. Overall, the practitioner determines the level of detail required depending on their specific application.

The RAPTOR software provides the most realistic representation of real-world modeling in the field of reliability simulation. Therefore, the capabilities of this software package define the basis for evaluating realistic implementation of RM&A behavior. The deficiencies in this software center around cost, maintenance practices and sparing strategies. Additional improvements to the software could include interdependencies,

preventive maintenance, priority of limited resources, and varying distributions for different maintenance categories. As mentioned earlier, no software can include all aspects of a real-world scenario, but increased details offers additional insight into reliability analysis.

Summary

Advancing the understanding of RM&A analysis remains the primary purpose of developing a robust simulation environment. Describing system performance, analyzing system behavior, and predicting output performance metrics improve as the state of reliability simulation improves. Easy to use, generic simulation tools capable of realistically representing real-world scenarios provide the practitioner an avenue for valuable insight into system behavior. Darren Durkee (1997) demonstrates the potential use of simulation results in a sensitivity study conducted as part of his Master's research. His study examines the affect on availability when the input distribution for a component's failure rate is changed.

The idea of sensitivity analysis is fundamental in improving reliability analysis, and the background comes from concepts in experimental design (Montgomery (1991)). Darren Durkee uses the RAPTOR simulation software for predicting performance of complex systems. He also cites Montgomery's work in the area of factor screening and implements similar concepts in his procedure. The procedure for performing sensitivity analysis includes steps in experimental design, simulation and output analysis. Similar analysis needs to be completed with respect to input assumption when little or no data is

available to determine the component's failure rate. A detailed study using a triangle distribution is provided in Chapter 4 which addresses these concerns.

Changing maintenance policies represent another area of research in which a robust simulation environment proves beneficial. A realistic simulation, allowing replace versus repair maintenance policies provides results for comparison of system behavior on availability, cost and other performance criteria. "Repair better than new" and "bad as old" maintenance policies are bound to effect output metrics. Furthermore, changes to software allow limited resource priorities to affect the order of repairs and affect system performance. All these changes increase the robustness of modeling real-world characteristics involving maintainability issues. The result is better understanding of system behavior and improved reliability analysis capabilities.

The shift in emphasis to reliability, maintainability, and availability in product and process improvements requires new criteria for decision making. An important consideration for management continues to be cost. Improvements in reliability simulation provide cost estimating capability along with continued standard output such as availability and logistic information. The question still remains, how is simulation implemented to assist in managerial decision making? A real example in trade-off analysis based on cost and supported with accurate and realistic output from simulation is provided in Chapter 4. The communications network identifies a procedure for comparing alternative sparing strategies.

A final area of improving reliability analysis rests in clarifying confusion in current literature. An efficient software package can be used to replicate previous results

in reliability analysis and demonstrate theoretical concepts developed in prior research. The time between failure research conducted by Barlow and Proschan (1965) is often confused and misapplied to reliability problems. Their findings are explained using output analysis and results provided from reliability simulation. The consequences of misusing the results can be detrimental in predicting system behavior. A detailed case study in Chapter 4 demonstrates the affect of erroneously applying the exponential time between failure distribution. Clarification of current research is fundamental to improving RM&A analysis using reliability simulation.

APPENDIX 2A

CONTINUOUS SIMULATION, SLAM LISTING

```
SUBROUTINE STATE

    COMMON/SCOM1/ATRIB(100),DD(100),DDL(100),DTNOW,II,MFA,MSTOP
,NCLR1,NCRDR,NPRNT,NNRUN,NNSET,NTAPE,SS(100),SSL(100),TNEXT,TNOW
,XX(100)

    SS(7) = INT(SS(1)*10000.0)
    DD(1) = 2.0*SS(2)
    DD(2) = -2.0*SS(2)+4*SS(3)
    DD(3) = -4.0*SS(3)+6*SS(4)
    DD(4) = -6.0*SS(4)+8*SS(5)
    DD(5) = -8.0*SS(5)+10*SS(6)
    DD(6) = -10.0*SS(6)

    RETURN
    END


GEN, CHAMBAL, PDE;

    CONTINUOUS,8,0,0.01,1.0,1.0;

    INTLC,SS(1)=0.0,SS(2)=0.0,SS(3)=0.0,SS(4)=0.0,SS(5)=0.0,SS(6)=1.0;

    RECORD,TNOW,TIME,0,B,1.0;

    VAR,SS(1),B,PO;

    INITIALIZE,0.0,2.0;
```

APPENDIX 2B

SLAM CODE FOR QUEUING SIMULATION

```
GEN, CHAMBAL, FIFI;

    LIM,1,3,5;

    TIMST, XX(10), AVAILABILITY;

    INITIALIZE, 0, 100;

    FIN;


SUBROUTINE EVENT(IX)

        COMMON/SCOM1/ATRIB(100),DD(100),DDL(100),DTNOW,II,MFA,MSTOP
,NCLNR

1,NCRDR,NPRNT,NNRUN,NNSET,NTAPE,SS(100),SSL(100),TNEXT,TNOW,XX(10
0)

    GOTO (3,6), IX

3 CALL FAILURE
RETURN

6 CALL REPAIR
RETURN

        END


        SUBROUTINE INTLC

        COMMON/SCOM1/ATRIB(100),DD(100),DDL(100),DTNOW,II,MFA,MSTOP
,NCLNR

1,NCRDR,NPRNT,NNRUN,NNSET,NTAPE,SS(100),SSL(100),TNEXT,TNOW,XX(10
0)

        CALL SCHDL(2,0.0,ATRIB)
```

```
      CALL SCHDL(2,0.01,ATRIB)
      CALL SCHDL(2,0.02,ATRIB)
      CALL SCHDL(2,0.03,ATRIB)
      CALL SCHDL(2,0.04,ATRIB)

      XX(1) = 1.0
      XX(2) = 2.0
      XX(3) = 3.0
      XX(4) = 4.0
      XX(5) = 5.0
      XX(6) = 0.5

      XX(7) = 0.0
      XX(8) = 1.0
      XX(10) = 0.0

RETURN
      END


SUBROUTINE REPAIR

      COMMON/SCOM1/ATRIB(100),DD(100),DDL(100),DTNOW,II,MFA,MSTOP
,NCLNR

1,NCRDR,NPRNT,NNRUN,NNSET,NTAPE,SS(100),SSL(100),TNEXT,TNOW,XX(10
0)

      IF (XX(8).GT.5.0) GOTO 20

      ATRIB(1) = TNOW
      ATRIB(2) = XX(INT(XX(8)))
      ATRIB(3) = XX(6)
      WRITE (NPRNT,10) ATRIB(2),ATRIB(3)
 10   FORMAT(F4.1," ",F4.1)
      XX(8)= XX(8)+1

20 ATRIB(1) = TNOW

IF (XX(10).GT.0.0) GOTO 30

XX(10) = 1.0

 30   XX(7) = XX(7) + 1
```

```
SCHDL(1,EXPON(ATRIB(3),9),ATRIB)

RETURN
END


SUBROUTINE FAILURE


COMMON/SCOM1/ATRIB(100),DD(100),DDL(100),DTNOW,II,MFA,MSTOP,NCLN
R

1,NCRDR,NPRNT,NNRUN,NNSET,NTAPE,SS(100),SSL(100),TNEXT,TNOW,XX(10
0)

XX(7) = XX(7) - 1.0

IF (XX(7).GT.0.5) GOTO 20

XX(10) = 0.0

20 SCHDL(2,EXPON(ATRIB(2),9,ATRIB)

RETURN
END


SUBROUTINE OTPUT


COMMON/SCOM1/ATRIB(100),DD(100),DDL(100),DTNOW,II,MFA,MSTOP,NCLN
R

1,NCRDR,NPRNT,NNRUN,NNSET,NTAPE,SS(100),SSL(100),TNEXT,TNOW,XX(10
0)

WRITE (NPRNT,10)  TTAVG(1)*100

10 FORMAT("AVAILABILITY  =  ",F5.2)

RETURN
END
```

CHAPTER 3

RELIABILITY SIMULATION

Introduction

After a thorough review of reliability simulation software, RAPTOR (Rapid Availability Prototyping for Testing Operational Readiness) offers the best basis for reliability software modification. All modifications provide additional insight into some aspect of system level reliability, maintainability, and availability analysis. The software is available at no cost from the public domain and can be acquired by contacting Ken Murphy at murphyk@afotec.com. The Air Force Test and Evaluation Center (AFOTEC) developed RAPTOR for the purpose of evaluating new systems. I took part in the test and evaluation of RAPTOR during the design and development phases of this process. The source code and simulation software (MODSIM II) was available to me as an Air Force member for my dissertation work. MODSIM II is sold commercially by CACI Products Company.

MODSIM II is an object-oriented simulation language based on the C++ programming language and includes a graphical user interface (GUI) builder (MODSIM II User's Manual, 1995). All RAPTOR modifications were made using the MODSIM II software. This object-oriented language provides a simple way to create nodes, blocks, and other constructs necessary for development of the reliability block diagrams found in RAPTOR. The blocks can inherit features from previously defined objects, along with additional parameters as specified. The MODSIM II program requires a version of

Visual C++ to be installed on the computer in order to run the system software. MODSIM II has a built-in debugger and allows for modular design of simulation code.

Raptor Review

The RAPTOR simulation software consists of ten modules. The following summary of these modules describes the flow of the simulation program. This review helps explain where the code modifications are made and how they affect the overall simulation program. Ten modules comprise the software program, a main program followed by nine sub-programs, each of which includes a definition and implementation file. (The nine sub-programs are rbdwin, Dialogs, FileN, Grid, RandomG, efpa, RBDRes, RBDBloc and engine.) The main procedure is called when RAPTOR is initially executed. This procedure begins a continuous loop and allows the user to interface with the program through the provided menu bars. This procedure was not modified by any code changes made to RAPTOR.

The rbdwin procedure handles the graphical user interface within RAPTOR. The main menus are defined here, along with the main window for the simulation software. Building the reliability block diagrams (RBDs) is a function of this procedure, as well as adding blocks, nodes and connectors. The editing functions, such as copy, cut and paste are also a function of the rbdwin procedure. Therefore, this procedure was modified extensively during this research to facilitate the inclusion of new features in the basic RAPTOR program. The definition and implementation files required changes to handle new features that allow advanced analysis into system-level reliability analysis.

The next module is the Dialogs procedure. This procedure handles the GUI dialog boxes, providing the user a method for inputting new data into the simulation program. All the RBD building is done through a point-and-click format, using pre-defined input dialog boxes to collect the appropriate information. As expected, new features added to the program require new input from the user; therefore, this procedure was modified when changes to the software were created and implemented into the code. It is important to remember that changes must be made to both the definition and implementation modules. The definition module allows the programmer to define the procedure and methods for each module, along with local variables.

The FileN module was affected by changes introduced to the RAPTOR simulation software. This procedure handles the file manipulation, including opening and closing files, reading and writing to storage files, and any other file management functions. The two main storage files are the filename.rbd file and the filename.def file. These two files include all the information about a given reliability block diagram, as well as the simulation options for running the analysis for the given system. All the block, node and connection information is saved so the user can re-open a saved file without having to recreate the entire system. All information collected through the Dialog module for the added features must be stored in these files to properly define the system. The text files (.rbd and .def) are modified, and the new files read and write the appropriate information as defined by the user.

The next four modules (Grid, RandomG, efpa, RBDRes) were not affected by code changes made to RAPTOR. The Grid module defines and displays the background

grid for the user while defining new system block diagrams. The RandomG module is responsible for generating random numbers during the simulation. These numbers are representative of the failure and repair distributions defined by the user. The efpa (Enhanced Failure Propagation Algorithm) procedure is used to determine the affect of failing each block in the defined system. The overall affect on the system can be discovered and annotated to speed up the actual simulation of the system. This entire procedure is completed before the actual beginning of the simulation. Finally, the RBDRes module handles all resources defined in RAPTOR, along with any pools defined by the user.

The RBDBloc module handles the graphical units used to build the reliability block diagrams. The blocks, nodes and connectors are manipulated using this module. As expected, changes to the software affected this module; the definition and implementation modules were edited to account for upgrades and modifications to the simulation software. Information defined by the user is passed to and from this module, the Dialog module, and the FileN module. These three modules continually interact with each other to develop the overall system RBD. Once all the appropriate information is collected and displayed, the system can be sent to the final module, the engine code.

The engine module is the core of the simulation program. All blocks are initialized and sent to this procedure to begin simulation. The blocks fail and are repaired while the effects of these changes are propagated throughout the system. The appropriate statistics are collected to provide output to the user upon completion of each simulation. This module will be discussed in more detail as changes are made to the simulation code

that directly affect how this procedure operates. The definition and implementation modules were updated to allow the basic simulation algorithm to behave differently, as represented by the new features added to the RAPTOR program. The heart of the simulation code rests in this module as these lines of code determine the actual system behavior.

Overview of Simulation Changes

The following section introduces the changes made to the simulation code directly affecting RAPTOR. All changes were made in an effort to provide additional features in the RAPTOR reliability analysis software. These additional features provide added insight into system analysis and offer the user a greater opportunity to understand their specific system behavior. The changes were made directly to the source code of RAPTOR and were integrated into a new executable software program with enhanced capabilities. Appendix 3A includes the source code files for the modification to RAPTOR.

The next sections breaks down each change to explain coding algorithms, module manipulation and the effect on simulation behavior. The core set of changes includes priority queuing, stream dependency, repair by adjustment only, reliability improvement/degradation, block level definition, an overall cost structure, and changes to emergency sparing for spare pools. Each change will be discussed individually and in detail. The purpose behind the desired modification, along with insight into how this change provides improved system analysis is also discussed. Finally, a simplified scenario is presented to highlight the improved simulation capability in a system analysis

case study. The scenarios compare output from previous RAPTOR simulations to the output from the modified RAPTOR program. Overall, the scenarios justify adding the given simulation features and provide insight into how these modifications can advance the area of system-level, reliability analysis.

ALGORITHM FOR PRIORITY QUEUING FOR LIMITED RESOURCES

Step 1 – Define RBD structure and use of limited resources (RAPTOR code).

Step 2 – Propagate component failure to determine effect on system status (RAPTOR code).

Step 3 – Assign higher repair priority for components causing system failure (new code).

Step 4 – Repair components using limited resources based on assigned priority (new code).

Step 5 – Return limited resources to the system for future use (RAPTOR code).

EXPLANATION OF PRIORITY QUEUING FOR LIMITED RESOURCES

RAPTOR has the ability to specify limited resources for maintenance of failed components. The limited resource may be manpower, maintenance equipment or any resource which must be shared among failed components. The current simulation software places a first in, first out (FIFO) priority on the component request for the limited resource. This does not take into account the effect the failing component has on the overall behavior of the system. When components fail, they are simply placed in the queue awaiting access to the limited resource.

There are many types of priorities for queuing structures such as FIFO, last in – first out, and priority based on importance (Law and Kelton, 1991). In the case of

reliability simulation, priority based on the system response to the component's failure is the most applicable. During the engine procedure for the simulation software, each component failure is propagated throughout the system. The component's failure causes additional failures downstream in the reliability block diagram. The failure propagates until reaching the stop node, or a redundant path that nullifies the effect of the failure. The redundant path performance is reduced or degraded; however, the system maintains its operating condition.

The propagation takes place in a subroutine called ReconfigureFail within the engine module. A priority value can be place on the individual component after examining the failure's effect on the system. A simple measure of the system status is used to determine the importance level for the access to limited resources. If the component's failure causes the system to fail, the priority is given a value of 2.0. If the component's failure does not cause the system to fail, the priority is given a value of 1.0. This guarantees that all components have their own priority value, and this value is based on the failure propagation. The components are then placed in the queue according to their priority for the limited resource.

The priority value is now used in the assignment of limited resources. During the operate subroutine, each block checks to see if limited resources are required. If they are required, the component must wait for the limited resource object to "PriorityGive" that resource to them before they can begin repair. The operate procedure gives the priority resource to the component and then takes the resource back upon repair completion. This affects the downtime of the system by increasing the priority given to components that

fail the system. The overall system downtime should decrease, while the availability should increase. The following case study highlights this concept and demonstrates the added utility of replacing the FIFO queuing strategy with a priority based on the importance of keeping the system in an operating condition.

CASE STUDY FOR PRIORITY QUEUING

A simple example is used to highlight the effect of having a priority queue placed on the limited resources. The reliability block diagram for the system depicted below is composed of a parallel and series structure.

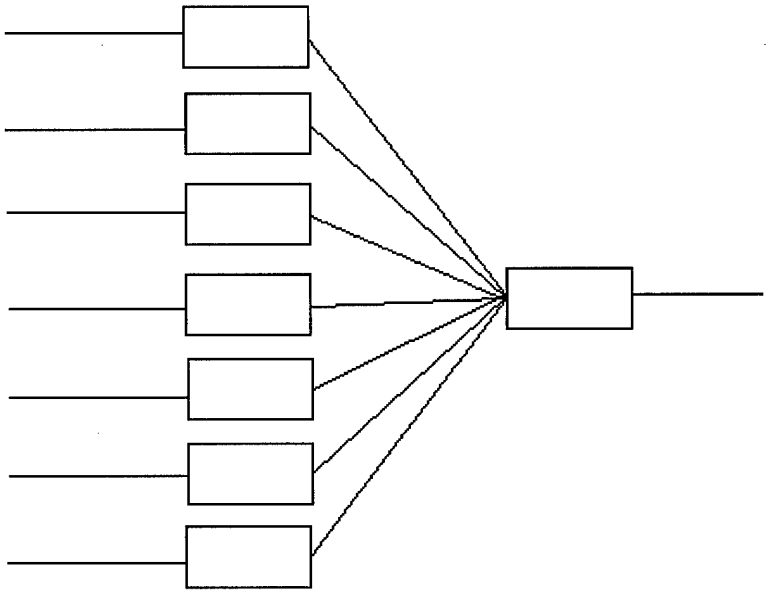


FIGURE 3-1. Reliability Block Diagram for Priority Queuing

There are seven components in parallel followed by one series component. All seven parallel components are identical; they have exponential failure distributions with mean 100 hours and lognormal repair distributions with mean 100 hours, and standard

deviation of 1 hour. The series block also has an exponential failure distribution with mean 100 hours, but has a lognormal repair distribution with mean 10 hours and standard deviation of 1 hour. The shorter repair times should increase the system availability.

The important factor in this simulation is the limited resource of manpower that all eight blocks require for repair. The parallel structure with the higher repair times is going to dominate the limited resource when no priority is given to the series block causing the system to fail. Only one of the seven parallel components is required to be operational in this system, and therefore, these components rarely fail the system. The series component has a shorter repair time but is stranded waiting for the limited resource being used by the parallel structure.

The system is simulated for 10000 hours to determine the overall system availability and the mean down time when the system fails. Both systems are simulated (with and without priority queuing) to compare the results of this scenario. The availability for the non-priority system is 26.0%, and the mean down time for the system is 296 hours. When priority is given to the series block that fails the system, the numbers are dramatically different. The new availability is 36.7% (an increase of over 40%), and the mean down time of the system decreases to 115 hours (a change of over 60%). This simple example demonstrates the impact of placing importance priority on limited resources used during reliability simulation. It is important to mention that no repair actions are preempted using this priority scheme. The entire repair action is completed and then the next repair action is chosen based on the remaining components in the queue.

ALGORITHM FOR STREAM DEPENDENCY FOR RELATED COMPONENTS

Step 1 – Define stream dependency while building the system RBD (new code).

Step 2 – Form stream dependent groups during initialization of the simulation (new code).

Step 3 – Determine if failed component is part of stream dependent group (new code).

Step 4 – Place dependent components in stand-by (new code).

Step 5 – Repair the failed component (RAPTOR code).

Step 6 – Notify all dependent components when repair is complete (new code).

Step 7 – Check if stand-by components should be operational, based on stream and system dependencies (new code).

Step 8 - Continue simulating the system until next failure (RAPTOR).

EXPLANATION OF STREAM DEPENDE NCY FOR RELATED COMPONENTS

The idea of stream dependency is very closely related to system dependency, which is a current capability of RAPTOR. System dependency allows components to enter a stand-by mode of operation while the system is in a failed state. This prevents additional hours of operation from accumulating on components until the system is returned to an up condition. Many components in a physical system are not required to be operational while the system is down. This feature simulates this situation. The component's life is therefore extended due to the period of stand-by operation. The system failure triggers a review of system dependent components and places them in stand-by until the system becomes operational.

Stream dependency is very similar to this modeling concept. The components are defined to be stream dependent instead of system dependent when they relate to other components in the system. For example, if there are five components in a series path and one component fails, the other four are stream dependent upon this failure and are placed in stand-by while the failed series component is repaired. The failed series path may or may not have taken the system down; however, the four other components of that path have no reason to continue operating if they can be temporarily shut down. These components are considered stream dependent, and no additional hours accumulate towards the failure of the components until the failed component is repaired.

The primary consideration when examining the idea of system and stream dependency is whether or not these related components can be physically turned off, or put into stand-by when the system or related components fail. A good example is a computer system. If the monitor fails, the CPU can be shut down so no additional hours accumulate on that component of the system. Once the monitor is repaired, the CPU can once again be turned on for operation. There are many times, however, when components can not be turned off or put into stand-by mode and must continue operating. In this case, they continue to accumulate hours and may fail while other dependent components are already in a down condition.

The first step in programming stream dependencies is to develop the input capability for the user to define related components. This is handled in the GetProperties subroutine of the Dialogs module. The user selects one of three stream dependent groups or defines each block as not stream dependent. The stream dependent choice is recorded

with each block and sent to the appropriate data structures to include this as a block parameter. The number of stream dependent groups is limited to three to demonstrate the concept of stream dependency. Additional dependency groups could be added by following similar coding changes and would increase the flexibility of the software. The user defines the stream dependency at the same time system dependency is declared. The basic block parameters are defined together during construction of the reliability block diagram. A component can be stream dependent and system dependent at the same time.

The next step in simulating stream dependency is to build those groups of components that are related to one another. During the Initialize subroutine of the Engine module, each component is questioned and added to the appropriate stream/system dependent group. The component can only be added to one stream dependent group defined by the user. The members of each group are manipulated during the simulation to control related components throughout the execution of the program. After each component failure, the simulation checks to see if the failed component is a member of a stream dependent group. This is handled by the ReconfigureFail subroutine of the Engine module. If this is true, all other members are interrupted and placed in stand-by mode. The dependent components wait for the failed component repair, and this action fires a trigger to return the stand-by components to operation. The ReconfigureRepair procedure of the Engine module handles the trigger release while the repair propagates throughout the system. Similarly, when the system fails, all system dependent components are placed in stand-by awaiting the repair of the overall system.

System and stream dependency is coded by interrupting wait commands given in the MODSIM II coding language. The interrupt allows these components to drop into separate algorithms based on their dependency. The more complicated part of the simulation is when a component is both system and stream dependent. A failed component places the entire stream dependent group in stand-by; however, this action may not fail the system. While the stream dependent group is waiting repair of the failed component, other conditions in the RBD may cause the overall system to fail. Therefore, before each stream dependent component can be taken off stand-by, they must check to see if the system is operational if they are system dependent. The simulation manipulates flag variables to identify which dependency group caused the stand-by operation. A nested while loop, found in the Operate procedure of the Engine module, controls the components status and return to the system. If the system is down, the components are placed in stand-by, waiting for the system trigger to fire.

The reverse does not happen, meaning system dependent components do not have to check if their stream dependent group is operational upon system repair. The system failure would have caused the stream dependent group to be placed in stand-by when the operation of the system dependent component was halted. Thus, no stream dependent components can fail while they are in stand-by waiting for the system to be returned to an operating condition. The system continues under normal operation after both dependency conditions are checked and the components return to operation.

CASE STUDY FOR STREAM DEPENDENCY

The following example highlights the importance of using stream dependency to reduce the failures associated with components accumulating unnecessary hours of operation. If the components can be physically placed in a stand-by mode when related components are waiting for repair, it is important to model this situation. The reliability block diagram shown below is used to demonstrate the effect of stream dependency.

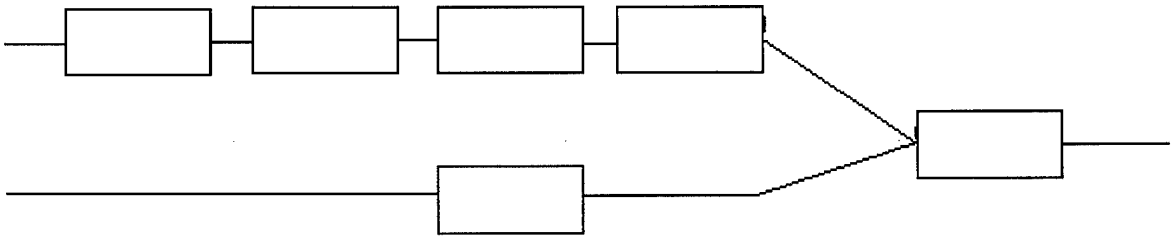


FIGURE 3-2. Reliability Block Diagram for Stream Dependency

The four components on the top path are stream dependent; if one component fails, the other three are placed in stand-by while the failed component is repaired.

The primary interest of this simple example is sparing. With the current version of simulation, the series path must remain operational at all times and thus experiences additional component failures. If in fact, the components can physically be turned off or placed in stand-by, this reduces the hours accumulating on these components leading to failures. The system remains operational as long as one out of the two paths is operating. This example highlights the new concept of stream dependency and more accurately models the sparing level required to support the system depicted above.

The four series blocks are identical with a exponential failure distribution with mean 100 hours. They are repaired according to a lognormal distribution with mean 100 and standard deviation of 1 hour. The single component on the secondary path has an exponential failure distribution with mean 1000 hours and lognormal repair distribution with mean 10 and standard deviation 1 hour. The downstream component that ties together both parallel paths has the same failure and repair distribution as the single block just described. The availability of the system is high due to the parameters of the final block and the single component on the redundant path. The scenario, however, is designed to model the effect of stream dependency on spare requirements for the four series blocks in the top parallel path.

The system is simulated for 10,000 hours to determine the number of spares required for the four series blocks. They are defined using a common spare pool since the components are identical and the spares can be shared amongst themselves. The first simulation involves no stream dependency, returns an availability of 98.6%, and requires the use of 229 spare components. A second simulation defines a stream dependent group including all four components. The availability is 97.6%, but only uses 103 spares. This is a direct result of the four series components being placed in stand-by while one of them is being repaired. The availability is not affected because of the parameters of the redundant path.

The sparing level required for this scenario varies a great deal dependent on the definition of the system. Accurately modeling stream dependency reduces the number of spares required by over 50%. This would dramatically impact the user's budget for

sparing allocation, storage and logistics. Although the structure and design of this case study is simplistic and developed to highlight the concept of stream dependency, it is easy to extrapolate this example to real-world scenarios with similar dependencies. The results may not be as dramatic, but modeling interdependent components is important in complex systems. Modeling components that can be physically shut down or placed in stand-by extends their operational life by reducing the accumulation of unnecessary hours of operation.

ALGORITHM FOR ENHANCE MAINTENANCE CAPABILITY TO HANDLE REPAIRS BY ADJUSTMENT AND REMOVE AND REPLACE SIMULTANEOUSLY

Step 1 – Define the remove and replace repair distribution (RAPTOR).

Step 2 – Define the repair by adjustment distribution (new code).

Step 3 – Define the probability of having one maintenance action versus the other, i.e., having a repair made by adjustment only versus using a remove and replace (RAPTOR).

Step 4 – When component fails, decide which repair action to implement (new code).

Step 5 – Increment the appropriate failure counters to monitor both maintenance actions (new code).

Step 6 – Calculate the time of repair based on the appropriate repair distribution and the appropriate failure counters (new code).

Step 7 – Upon completion of repair, activate the component (RAPTOR).

EXPLANATION OF ENHANCED MAINTENANCE CAPABILITY TO HANDLE REPAIRS BY ADJUSTMENT AND REMOVE AND REPLACE SIMULTANEOUSLY

The current version of RAPTOR is capable of handling repairs made by adjustment or repairs using a remove and replace concept. The key issue is handling both maintenance procedures simultaneously. If the user defines an infinite number of spares, they are representing maintenance or repair by adjustment. When a component fails, the simulation always finds an available spare, thus, the repair by adjustment can be completed. If the spares are limited, the repair action is a remove and replace, and the simulation must first query to determine if a spare is available. An infinite number of spares can be defined for remove and replace repair procedures, but this makes little sense in determining the system behavior (unless the user is simply concerned with determining spare requirements).

There are many systems in which both maintenance actions take place on the same component. There may be times when the failed component is simply repaired (with no need for additional spares), while at other times the component must be removed and replaced with an identical spare. This situation leads to the modifications and code changes allowing the user to specify both maintenance actions on the same component. Separate distributions are specified for both actions and can be completely different in terms of distribution choice and parameter values. Additionally, the user specifies the probability of using one maintenance action over the other. As an example, a component may be repaired by adjustment only 80% of the time. The repair follows a lognormal distribution with a mean of 10 hours and standard deviation of 1 hour. The other 20% of the time, the component is simply removed and replaced with an identical spare. This repair action follows a normal distribution with mean 4 hours and standard deviation of

10 minutes. The remove and replace maintenance is much quicker and experiences a smaller degree of variation.

All the information pertaining to a given block is stored in the appropriate data structure. The Dialog module accepts the user input from the user interface to initially store the parameters into their locations. The parameters for both distributions are kept for use during the course of the system simulation. The FileN module handles the storage and retrieval process when the system is loaded and closed in the software. Additional parameters to track statistics are defined to handle the increased maintenance capability. Mean time to repair for both actions are monitored and reported separately and as a joint statistic. Finally, failure counters are added to monitor which repair actions are being implemented. (These counter values directly impact the next failure time value calculated by the software when reliability degradation/improvement are included.)

The repair by adjustment code mirrors the original repair coding found in RAPTOR. Validation checks are made to determine if the user has entered valid parameters for the specified distribution. The same dialog boxes and distribution choices are presented, but now in duplicate to handle both maintenance operations. Fundamental differences are highlighted as the system begins simulation and the components begin to fail. When the component reaches its time of failure, a uniform random number is drawn and compared to the specified probability of performing each maintenance action. This tells the simulation which distribution to use to calculate the component repair time. The appropriate failure counter is incremented to track the number of repairs by adjustment and repairs by remove and replace.

A repair by remove and replace must request the presence of a spare component before continuing with the repair action. Repair by adjustment, however, skips this step and does not check the sparing status. This also gives the user a chance to analyze sparing requirements for specialized maintenance including both repair actions. After the component waits the required administrative and logistics delay time, the repair action takes place. The code is identical for both repair actions, just the distribution and the parameters listed in the function call are changed to reflect the appropriate repair. The component is delayed for the calculated repair time and then returns to the system in an operational condition. All statistical monitors are updated and reported at the end of the simulation. The user is able to analyze both maintenance actions and the entire maintenance concept as a whole. The probability of having one maintenance action versus the other can be manipulated to observe the system impact. This idea is highlighted by the case study in the next section.

CASE STUDY FOR MULTIPLE MAINENANCE CAPABILITY

The purpose of this case study is to demonstrate the advantage of offering multiple maintenance strategies for a single block. Each block can be defined with two repair distributions, one for remove and replace actions and one for repair by adjustment actions (no spares required). The user also defines the probability or proportion of the time in which the maintenance actions take place. This probability is varied from zero to one hundred percent to demonstrate the effect on the overall system performance. The reliability block diagram shown below describes the structure of the system under test.

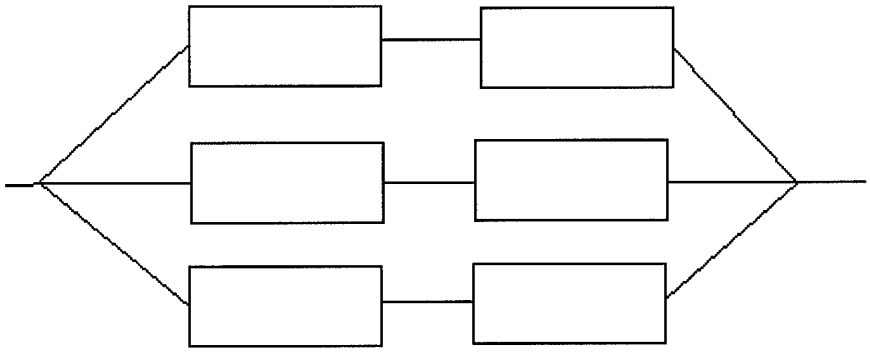


FIGURE 3-3. Reliability Block Diagram for Multiple Maintenance Strategies

There are three parallel paths, each with two blocks in series. The failure rate for all six identical blocks is exponential with a mean of 100 hrs. The remove and replace maintenance distribution is lognormal with mean 10 and standard deviation of 1 hour. The repair by adjustment distribution is also lognormal, but with mean 25 and standard deviation 2.5 hours. The remove and replace actions are quicker to perform, as expected, than the repair by adjustment actions.

The parameter of interest is the probability of having one maintenance action versus the other. To begin this case study, the probability of using remove and replace is set to 100%. This model behaves exactly like the original RAPTOR simulation code. The system has an infinite number of spares for each block. Next, the probability of having a remove and replace action is reduced in 25% increments until all maintenance actions become repair by adjustment only. No spares are required for these actions;

however, the system does experience a longer distribution for repair times. The system is simulated for 10,000 hours to determine the overall availability, mean repair time and the total number of spares required. This information is shown in the table below.

% R&R	100	75	50	25	0
Availability	93.1	86.2	78.9	74.0	69.4
R&R MRT	10.0	10.0	9.97	10.0	0.0
Rep MRT	0.0	25.1	24.8	24.8	25.0
Avg MRT	10.0	13.8	17.5	21.0	25.0
Spares	533	385	269	133	0

TABLE 3-1. Output for Multiple Maintenance Example

The mean repair times (MRT) are equal to the mean for the appropriate distributions, as expected with the high number of failures experienced by the system (greater than 100). The average MRT, which is based on both maintenance actions, is an average of both distributions. The average MRT is biased by the distribution from the higher percentage maintenance actions. As an example, 75% remove and replace and 25% repair by adjustment return an average MRT of 13.8 hours. This is very close to the expected value of 13.75 hours (.75*10 + .25*25).

The spared required by the system decreases as more and more maintenance actions are performed by adjustment only. The trade-off lies in the availability of the system. Since repairs by adjustment require significantly more time (25 versus 10 hours),

the availability drops from 93% to 69%. The user must analyze this information with relation to the given system. The loss of availability may be acceptable with the monetary gain from reduced spare cost. Most likely, there would be some middle ground, where the system experiences some remove and replace actions to keep availability at a higher level. The user may not have control over the percentage of remove and replace actions versus repair by adjustment. This probability may be a result of data collection over time, monitoring the types of maintenance actions completed on the system.

This case study clearly shows how changing the maintenance concept for a given system can directly impact the availability and sparing requirements. The user gains additional insight into system behavior as the maintenance strategy is manipulated through simulation. There are times, however, when the maintenance concept stems from the physical system. Many components or subsystems are "black boxes" and can only be repaired by swapping them for identical spares. In this case, the user may be modeling the system to determine the benefit of acquiring a repair capability for the "black boxes". Multiple maintenance actions for individual blocks are very realistic and are present in many repairable systems. This feature adds flexibility to the modeling capability of RAPTOR.

ALGORITHM FOR RELIABILITY IMPROVEMENT/DEGRADATION

Step 1 – Define the improvement/degradation factor repair and remove and replace (new code).

Step 2 – Define the improvement/degradation limitation repair and remove and replace (new code).

Step 3 – Maintain failure counters for repair and remove and replace maintenance actions (new code).

Step 4 – Repair failed items according to appropriate distribution (new code).

Step 5 – Determine next time to failure modified to reflect reliability improvement/degradation. This incorporates the failure counters, improvement/degradation factor and improvement/degradation limit (new code).

Step 6 – Continue simulation using failure times and collect appropriate statistics (RAPTOR).

EXPLANATION OF RELIABILITY IMPROVEMENT/DEGRADATION

The current version of RAPTOR requires all repairs to be "good as new," meaning the next time to failure is based on the same distribution as all failure times. There are many situations in which this does not hold true. Often, repairs made by adjustment do not return the component to a new condition, rather the component is somewhat degraded and has a higher failure rate. Conversely, there are situations when the repair may actually improve the system by increasing the reliability of the repaired component. A defected component, when repaired, may actually have a lower failure rate and now become "better than new." The entire repair process can affect components in a positive or negative way. Components can be "better than new," "good as new" or "bad as old."

The reliability improvement/degradation algorithm allows the user to control repair process. The user can specify, using a control factor, whether the repair of the component increases or decreases the next time to failure. Additionally, a limit to the reliability improvement/degradation is included to prevent the reliability from increasing linearly too high, or decreasing exponentially too low. The factor and limitation values are specified for both repair by adjustment and remove and replace maintenance strategies. One repair action can improve the reliability (i.e., the remove and replace), while the other action can degrade the reliability (i.e., the repair by adjustment). The failure counters maintain the relationship between repairs and remove and replace and are used to calculate the appropriate time to next failure.

The user specifies all four variables through the Dialogs module. New parameters are defined for every block to capture the user input and store these values in the appropriate locations. Four values are requested as the user specifies the failure and repair distributions for each component. The GetDistro procedure displays the dialog box and stores the parameter values when the dialog box is closed. A value of 1.00 is entered as the improvement/degradation factor to model "good as new" behavior. These values must also be store externally as part of the structure file to be retrieved when the system is reloaded into the RAPTOR software. The FileN module handles storage and retrieval of saved systems.

The simulation runs without change until a component fails. The code is modified to track the total number of failures, along with the number handled by each type of maintenance action (repair versus remove and replace). These counters are found

in the Operate procedure of the Engine module. The counters are used later when determining the time to next failure. The repair action takes place according to the user-specified distribution. The code continues until the repair action is complete. The software must calculate the next failure time for the given component. All reliability improvement/degradation factors and limitations, along with the failure counters are incorporated into this calculation. The simulation examines ten possible scenarios to determine what equation to use to calculate the time to next failure.

A nested IF/ELSE IF structure is used to manipulate the time to failure. The Operate procedure is modified to include the appropriate code. The first time through the procedure, the component has not failed; thus, the time to next failure is not modified. At start-up the first failure times are calculated. The second and remaining times the components fail, they are categorized by the IF/ELSE IF structure. The same component can experience degradation and reliability improvement based on the repair action. The ten scenarios for calculating the next time to failure are: 1) No failures, 2) No reliability improvement/degradation factor for either maintenance action, 3) Remove and replace (R&R) reliability improvement with no repair by adjust improvement/degradation factor, 4) Remove and replace degradation with no repair by adjustment factor, 5) Repair reliability improvement with no R&R factor, 6) Repair degradation with no R&R factor, 7) Remove and replace is improving the system and repair is also improving the system, 8) Remove and replace is improving the system and repair is degrading the system, 9) R&R is degrading the system and repair is improving the system, 10) R&R and repair are both degrading the system.

All ten situations are not discussed separately, but the code is shown in the appendix. One example is given to demonstrate the impact of reliability improvement/degradation and limiting factors. Reliability improvement improves the system linearly, while degradation is handled in an exponential fashion. Assume a component is defined with both remove and replace and repair by adjustment distributions. The remove and replace action improves the reliability of the system ("better than new") while the repair action degrades the system ("bad as old"). Both parameters have a limitation to prevent the improvement/degradation factors from becoming too excessive. The formula to determine the next failure time is given by the following code:

```
ELSIF (RRAdjfactor>1.0) AND (RAdjfactor<1.0)

TempFactor := (((RRAdjfactor-
1.0)*FLOAT(RRFailureCounter))+1.0);
IF TempFactor > RRLimfactor
   TimeToFail := RRLimfactor*TimeToFail;
ELSE
   TimeToFail := TempFactor*TimeToFail;
END IF;

IF (RFailureCounter=0)
   TimeToFail := TimeToFail;
ELSE
   TempFactor := 1.0;
FOR j := 1 TO RFailureCounter
   TempFactor := RAdjfactor*TempFactor;
END FOR;
IF TempFactor < RLimfactor
   TimeToFail := RLimfactor*TimeToFail;
ELSE
   TimeToFail := TempFactor*TimeToFail;
END IF;
END IF;
```

```
{Handles the case when remove and replace is improving the
system, and repair is degrading the system.}
```

The failure counters control the amount of improvement and degradation by maintaining an accurate count of past maintenance actions. The repair counter (RfailureCounter) is reset to zero after every remove and replace action. This assumes the remove and replace action uses a spare component which has not experienced any repair actions, hence the counter is reset to zero. If the remove and replace counter (RRFailureCounter) is never resest, new spares continue to improve the system unless controlled by the limitation factor. The first portion of the code manipulates the times to next failure based on the reliability improvement experienced when using remove and replace. Next, the time to next failure is readjusted taking into consideration the number of repair actions competed since the last R&R took place. Finally, both modifications are compared against their respective limiting factors to ensure they do not exceed this threshold. The time to next failure (TimeToFail) is returned, and the component failure is added to the event calendar.

The simulation code continues as designed once the next time to failure is calculated. This feature adds flexibility to block definitions. If the reliability improvement/degradation factor and limit are set to the same number, the algorithm models a component with spares from a separate population. The block can be defined, and after the first failure, all subsequent components experience a different type of failure distribution. The new distribution is characterized by the same type but experiences a higher or lower mean. The component can not return to the original distribution once the

first failure occurs. The case study given in the following section is used to highlight the advantage/flexibility of using a reliability improvement/degradation factor. For simplicity sake, the repair strategy is strictly remove and replace. No repair by adjustment actions are considered.

CASE STUDY FOR RELIABILITY IMPROVEMENT/DEGRADATION

The purpose of this case study is to highlight the advantage of using reliability improvement to accurately predict system behavior. The repair actions are limited to remove and replace, and it is assumed the components experience reliability improvement. The spares used to repair the system have a higher reliability than the original components. The time to failure of the components is extremely high, hence, the replacement components have improved due to advances in technology. The reliability block diagram given below describes the structure of the system.

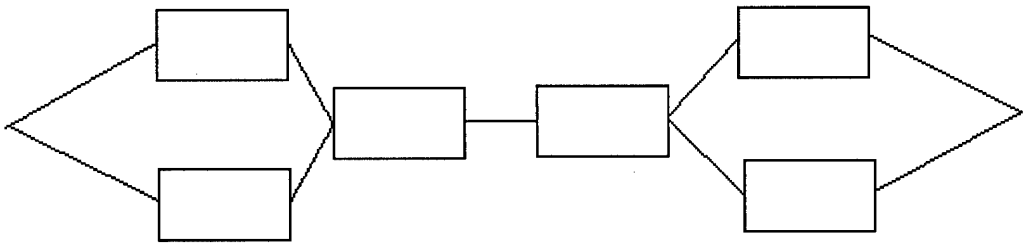


FIGURE 3-4. Reliability Block Diagram for Reliability Improvement/Degradation

The system is composed of series and parallel components. This system can be solved analytically, and these calculations are used to evaluate the simulation results.

All components are identical with an exponential failure distribution (mean equal to 90 hours) and a lognormal repair rate (mean equal to 10 hours and standard deviation 1 hour). The analytical reliability of this system is 79.39%. Modeling this system as event blocks, a feature of RAPTOR, the reliability is equal to 79.56 % after 10,000 simulations. The system can also be modeled to determine the steady-state availability that is equal to the reliability. After 100,000 hours of simulation, the availability is equal to 79.38%. This shows the accuracy of modeling system performance using RAPTOR simulation software. The system changes when the components are no longer repaired "good as new."

The case study changes to take into account a repair action that reduces the reliability of each component. The repaired component is "bad as old," or experiences a reliability degradation over time. The degradation factor is set equal to 90% with a limit of 50%. This implies the component failure rate increase after every repair. The original exponential mean for the failure distribution is 90 hours; this decreases to 45 hours. Hence, the reliability of the system decreases to take into account the inadequate repair capability experienced by the system. The degradation factor affects all components, series and parallel and drives the availability of the system lower.

The system should approach an analytical solution of 62.59%. This value is found by setting the component reliabilities equal to 81.82%, the lower limit of the reliability degradation. The simulation runs for 100,000 hours and returns an overall availability of 63.58%. As expected, this figure is biased by the initial time period before the degradation reaches the lower limit. The user is able to see the transition from high to

low reliability as the degradation factor decreases the effect of the repair action. The repair action never returns the components to "as good as new" condition, yet never degrades them lower than 50% of their original value.

The case study can also examine the "better than new" replacement capability. The parallel components are replaced with improved spares upon failure. The improvement factor is set equal to 120% and a limit of 200%. This provides an increase in system availability due to the improved parallel components. Their mean failure rate increases from 90 hours to 180 hours and thus the new analytical solution is 89.25% availability. The improvements affect all components equally, series and parallel. The simulation runs for 100,000 hours and returns an availability of 89.95%. This demonstrates the reliability improvement factor and how the overall system availability is positively affected.

This feature could be used to represent changing spare populations. The user could define the components to have "better than new" replacement capability due to an improved spare population, while the components are repaired to be "bad as old." The "bad as old" repair could be from poor maintenance practices. In this case, the improvement/degradation and limit factors are set to the same value. The replacement parts experience a lower rate of failure while repair parts experience a higher rate of failure. This affects the reliability and availability of the system. The user sets multiple maintenance strategies to model their system. The components are repaired 50% of the time, and use remove and replace maintenance the other 50%. The repair-by-adjustment action degrades the reliability, but the remove-and-replace improves each component

better than its original status. The lower limit for degradation is rarely attained. The remove and replace takes place prior to the degradation from repair by adjustment decreasing the reliability to the full extent. It would take 7 repair actions with no remove and replace for the component reliability to hit the lower limit (.9^7=.48, thus .50 lower limit is reached).

This scenario is simulated to determine the effect on the system availability. The availability should be between 89.25% and 62.59%. (These values are given above and represent upper and lower limits to the system availability.) The simulation algorithm resets the component reliabilities after each remove and replace maintenance action adjusted by the proper improvement factor. The components degrade during repair actions and cause the system performance to decrease. The simulation ran for 100,000 hours and returned an overall availability of 68.48%. This case study demonstrates the flexibility of the additional features in RAPTOR and also shows how the features can be used simultaneously to better model a real-world system.

The table below summarizes the case study and demonstrates the utility of these software changes. All components are identical and experience the same improvement and degradation influence. The system availability is the primary metric of interest and can be compared to the true value found analytically. In real-world situations, the analytical solution may be impossible or difficult to compute, therefore, this provides support for the accuracy of the simulation.

Comp Reliability	Improve Factor	Improve Limit	Degrade Factor	Degrade Limit	True Availability	Simulated Availability
90%	1	1	1	1	79.4%	79.6%
90%	1	1	.9	.5	62.6%	63.6%
90%	1.2	2.0	1	1	89.3%	90.0%
90%	1.2	2.0	.9	.5	62.6-89.3%	** 68.5%

** 50% repair actions and 50% remove and replace actions.

TABLE 3-2. Output for Reliability Improvement/Degradation Example

ALGORITHM FOR LEVEL DEFINITION FOR BLOCK STRUCTURE

(This coding change applies only to the Motorola case study provided.)

Step 1 – User defines the appropriate level for each block in structure (new code).

Step 2 – During event block modeling, down systems report the number of failures per level based on user input (new code).

Step 3 – The number of failures is output to a file for user analysis (new code).

Step 4 – The counting variables are reset to zero at the start of each additional simulation run (new code).

EXPLANATION OF LEVEL DEFINITION FOR BLOCK STRUCTURE

(These coding changes are not generic and only work for this one particular case study. They do not fit other level-definition problems.)

This software modification is explicitly added to accommodate evaluation of the detailed case study following this discussion. The intent is to demonstrate the feasibility of directly modifying generic code for specific applications. The current RAPTOR software does not allow the user to define a block level during the construction of the reliability block diagram. The user can not determine which level of the RBD is causing system failure. This enhancement assumes the user is conducting event block modeling therefore determining the system status at simulation end time is appropriate. The user is concerned with improving the overall system but is focused on spending additional money to improve the level of components causing the most system failures. The software changes are explicitly related to the case study and are incorporated with the given reliability block diagram. These block level definition changes will not work correctly with other reliability block diagrams. The failure criteria included in the code definition for the output report would be different for any other system.

The user specifies the level (one through five) while defining the components of the RBD. The dialog boxes are modified, along with the passing of parameters among the RBDBlock, RBDWin and FileN modules. The level variable is also stored in the output files to allow for reconstruction of systems previously developed. The number of levels is restricted to five to demonstrate the utility of this improvement and the added flexibility in modeling system behavior. The code changes can be easily modified to accommodate additional levels if the user determines this to be necessary.

This software change assumes the user is performing event block modeling. Operating blocks can be used, as long as the user is evaluating mission reliability.

Mission reliability is the reliability after a pre-determined number of operational hours. At this point, this feature identifies the block level responsible for the mission failure of the system. The case study analyzes this situation in a complex communications network. The levels represent separate equipment in the communications network of the physical system. They are explicitly seen in the construction of the reliability block diagram. The communications network is simulated for a period of time, and the user is interested in which level, or component type, is causing system failure. The appropriate number of component failures per level is reported and changes can be made to the system structure. This provides the user insight into which components should be improved to improve overall performance.

The Engine module is modified to track level failure during the EndOfRunUpdate procedure. The counting variables for each level are incremented when the system ends in a red, or down state. These counting variables are compared against the number of components required to determine which level caused system failure. Although the component failures are propagated downstream, they do not cause those components to fail. The system is degraded as designed by evaluating the K out of N requirements at the specified nodes. Therefore, the number of failures in each level is not affected by other failures. The output report is generated for each system failure and saved to a default file for further inspection by the user.

CASE STUDY FOR LEVEL DEFINITION

The case study demonstrates the potential "what if" analysis available when the reliability block structure is modified to include level definitions. This case study also

verifies the flexibility in changing the software for a specific purpose when the user is familiar with the coding structure. The levels are limited to five as required by the communication scenario in this example. The output report provides level analysis, allowing the user to allocate additional resources to the most productive area. The reliability diagram below represents a complex communication network. The components are redundant and system availability is the primary concern.

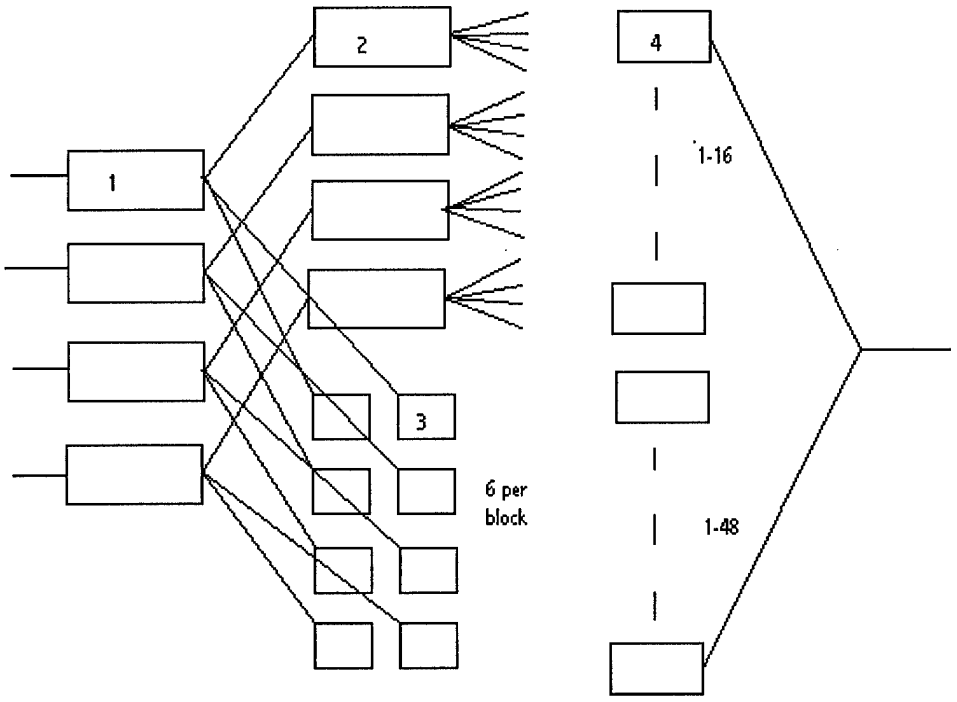


FIGURE 3-5. Reliability Block Diagram for Level Definition

The complex communications structure depends on redundancy to provide high reliability. The sixteen upper units are in parallel, and the system demands twelve of these paths to be operational. Of the 48 lower channels, 40 must be operational for the

system to be functioning. The relays and switching equipment cause multiple path failures and can fail the system. Overall, a five-month reliability is analyzed and level definitions are given for each of the component types (one through five). The component type reliabilities are as follows: 4 type-one at .99302, 4 type-two at .98874, 8 type-three at .98874, 16 type-four at .98268, and 48 type-five at .98268.

The original reliabilities are used to simulate the system and predict the five-month reliability using 10,000 simulation runs. Event blocks are used for all blocks. The system fails due to one or more level one failure, two or more level two failures, two or more level three failures, five or more level four failures, nine or more level five failures, or a combination of component failures from different levels. The five-month reliability is .9576 with the system experiencing 424 failures. The level definitions identify which component types cause the majority of these failures and where additional resources would generate the most improvement. The reliability of the level one, two and three components is manipulated to analyze the effect on the system performance. The reliability block diagram structure remains the same.

The number of system failures decreases and represents the impact of improving component reliability. The improved reliabilities can be analyzed to determine which component type (as defined by levels) offers the greatest benefit to system performance. The scenario assumes reliability improvements can be made in all components. The cost of increasing a component's reliability changes for each level in the system. Increasing level one component reliability from .99302 to .995 has the same assumed costs as increasing level two and three component reliability from .98874 to .993. This provides

the user a common cost constraint to identify which improvement provides the greatest benefit.

The table given below accounts for the system failures on a per level basis. Combination one (C1) failures are due to one level 2 failure and one or more level four failures. Combination two (C2) failures are due to one level 3 failure and three or more level five failures.

Component reliability			Level responsible for system failures						
1	2	3	1	2	3	4	5	C1	C2
.99302	.98874	.98874	284	30	0	5	0	31	78
.995	.98874	.98874	211	30	0	5	0	31	78
.99302	.993	.993	284	11	0	1	0	21	47
.995	.993	.993	211	11	0	1	0	21	47

TABLE 3-3. Output for Level Definition Example

The total number of system failures decreases as changes are made to the component reliabilities. The original system experienced 424 failures, although the table indicates 428 failures. There are four duplicate conditions causing failure to the system. For this analysis, both duplicate failure modes are considered relevant. The primary level causing these failures is level one, representing 66% (284/428). The level one reliability is increased, and the number of failure drops to 211. A 17% decrease in system failures

occurs due to the reduction in level one failures. It is interesting to note that the combination failures remain constant. This indicates that these failures are due to the level two and three components.

The level two and three reliabilities are changed from .98874 to .993, and the level one reliabilities are returned to their original value. The number of system failures decreases by 15%. In this case, the combination failures show a reduction in causing system failures. Assuming these two reliability improvements cost the same, the user chooses to increase the level one reliability for higher overall system performance. The system experiences a lower number of failures (355 versus 364), and the use of additional resources is optimized.

Finally, both level reliabilities are increased, and the total number of system failures is decreased to 291. This defines the peak system performance if enough resources were available to make reliability improvements in all three levels. The structure of the RBD remains the same, and the five-month system reliability increases to 97%. Further studies could analyze the effect of increasing the redundancy in the system. The physical system would have to be considered to determine if this modification is feasible. The level four and five components were not changed due to their limited impact on the system performance. However, they would be included if the redundancy aspect of the scenario were investigated.

ALGORITHM FOR COST STRUCTURE: OPERATIONAL COST

Step 1 – Define the operational cost factors, to include up cost, down cost, stand-by cost for the blocks and nodes in the reliability block diagram (new code).

Step 2 – Define new statistical tracking parameters to determine node and block cost during the simulation run (new code).

Step 3 – Maintain status of node and blocks to determine the appropriate calculation to cumulate node and block cost (new code).

Step 4 – Calculate node and block cost during simulation based on input parameters (new code).

Step 5 – Output node and block cost at the end of each simulation run (new code).

EXPLANATION OF OPERATIONAL COST STRUCTURE

The current version of RAPTOR has no means for calculating the cost of performance for a given system. The key output parameters are availability, maintainability and reliability statistics. The software does not consider cost to be an output metric of concern. There are many systems in which cost is the driving factor in development and design, operations and overall mission success. The software is modified to include an overall cost structure, providing the user with additional metrics to evaluate system performance.

Operations costs include up cost, down cost and stand-by (green, red and yellow) cost for blocks and nodes defined in the RBD. These factors are defined for each component and can help the user approximate the cost of operating a system over time. These costs can also be production values. Instead of cost over time, the system can be evaluated on output over time. The green, yellow and red represent the full, partial and halted capacity of a system or subsystem. Each block can represent a machine capable of

producing so many parts per hour. When the block is up, the system tracks the output from each component and returns a total metric after each simulation run.

The dialog boxes are modified to allow the user to define the desired parameters. All unused variables are set equal to zero to nullify their affect on the system. The Dialog module is modified in the GetParameters procedure. The RBDBlock module is modified in the SetBlockData procedure. And finally, the RBDwin module is modified in the GetBlockName, CopyBlock and PasteBlock procedures. These parameters are stored in the appropriate locations and shared among the modules to help implement the operations cost structure.

Once the modules above are updated, the Engine module can be manipulated to track the system cost for operations. Each block status change is registered, and the appropriate cost factor is used to increment the overall block or node cost. The cost is calculated by multiplying the time spent in each operation condition by the respective operations factor (i.e., two hours of up time multiplied by $10/ hour for up time is $20). The status-tracking variable is updated and set equal to the current simulation time. The cost statistics are continuously tracked and accumulate during each simulation run. At the end of each run, the total block and node cost are output to the appropriate cost files.

The stand-by cost for blocks can be considered an operations cost, or maintenance cost included in the next section. The stand-by cost becomes a factor when the components are system or stream dependent. The interrupt command in MODSIM places the components in a while loop, waiting for the related components or the system to become operational. During this time, the components accumulate hours adding to the

total cost of operating the block. The stand-by cost is more than likely less than the operating (green) cost but cheaper than the failed (red) cost. The red cost is more influential during the maintenance cost as seen in the section following the case study.

The node cost is modeled in the same manner as the block components. The green, yellow and red costs input by the user represent full, partial and zero operational capability. The costs are calculated based on the changing status of the nodes within the system. The K out of N defined by the user determines the condition of each node. If all the paths leading into the node are operational, the node is green. If there are more paths (greater than K) than required with one or more down paths, the node is yellow. If the number of operational paths leading into the node drop below the number required, the node is red. The status of each node is tracked and used to calculate the respective costs. Node costs can is easily interpreted as production quantity. Similar to using blocks as production per component, the nodes represent production for an assembly line or manufacturing unit composed of many components or machines (modeled by the blocks defined in the RBD).

CASE STUDY FOR OPERATIONS COST MODELED AS PRODUCTION

The case study demonstrates the potential for using operations cost to model production capacity of a given system. The reliability block diagram is defined with cost parameters for all nodes defined to represent production. There is full and partial capacity based on the status of the individual nodes. The system is defined below and described by the given diagram.

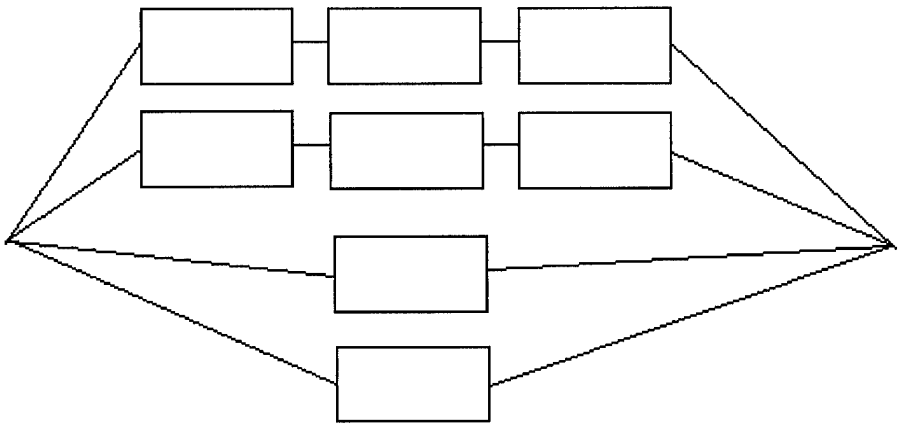


FIGURE 3-6. Reliability Block Diagram for Modeling Production

The two four paths represent production capability for a manufacturing setting. The six components in the top two paths have exponential time to failure with a mean of 200 hours. The repair rate is lognormal with mean and standard deviation equal to 10 hours and 1 hour, respectively. The bottom two blocks have exponential failure with mean 100 hours with the same repair distribution as the other blocks. The key to defining this system is the production capacity. The top line produces 100 widgets per hour. The second line is slower and only produces 50 widgets per hour. This could be due to personnel, equipment or other manufacturing conditions. Both bottom lines produce 120 widgets per hour. These could represent newer equipment that does the same job as the three components previously used. The failure rate is higher but the production capacity offsets this value.

The simulation runs for 100,000 hours, tracking the number of widgets produced. The average hourly production is reported and can be used to analyze the system. Using the reliability of each path, the analytical production for this simple system is approximately 348 widegets per hour. The simulation returns a production capacity of 347.92 widegets per hour, precisely what is expected. The justification behind this case study is to give confidence to the simulation in a simple design. Once the user is comfortable with running the simulation software, complex systems can be modeled and analyzed. The more detailed systems include administrative and logistic delays, system and stream dependence, multiple maintenance strategies and unique sparing considerations. These systems can not be solved analytically, and the improved software is capable of reporting a system-level production capacity.

The operations cost can be modeled directly instead of assuming the cost factors represent production. In this case, the user is able to model system cost and use these figures to make improvements to the system. A "what-if" analysis can be done to compare alternative configurations, reliability specification and RBD structures. The system can be compared on the basis of operational cost using the block and node cost definitions. The operations cost can be merged with maintenance cost. Maintainability cost is discussed in the next section and explained individually. However, combining both cost structures provides the user with an overall picture for system cost that includes operations and maintenance.

ALGORITHM FOR COST STRUCTURE: MAINTAINABILITY COST

Step 1 – Define the operational cost factors, to include repair cost for both maintenance strategies, administrative and logistic delay cost, sparing cost and initial cost for the blocks in the reliability block diagram (new code).

Step 2 – Define new statistical tracking parameters to determine block cost during the simulation run (new code).

Step 3 – Maintain status of node and blocks to determine the appropriate calculation to cumulate block cost (new code).

Step 4 – Calculate block cost during simulation based on input parameters (new code).

Step 5 – Output and block cost at the end of each simulation run (new code).

Step 6 – Reset the cost parameters to zero during the initialization of the simulation start time (new code).

EXPLANATION OF MAINTAINABILITY COST

This section is very similar to the previous section that included operational cost. The details are handled separately to provide a better explanation of the parameters included in maintainability cost. The current version of RAPTOR does not include any cost parameters. The upgraded/improved software uses the structure in place to add statistical parameters to calculate maintenance cost. The engine running the software is not changed by these modifications; RAPTOR is improved by providing the user with an additional method for analyzing system performance.

The input and user interface is handled through dialog boxes and manipulated in the Dialog module. The parameters are stored and retrieved through the FileN module as each additional variable must be stored in the proper description file to allow the user to access previously defined systems. The RBDwin module and RBDBlock modules are also affected as the parameters are exchanged between procedures and passed back and forth within the simulation. The added parameters include an initial block cost, sparing cost, repair cost for both maintenance strategies, and an administrative and logistics delay (ALDT) cost. The current version of RAPTOR includes the base parameters, and the modifications simply add cost parameters correlated to the input requested by the user.

The initial block cost and spare cost are kept separate to give the user more flexibility in specifying block cost. The initial cost is simply accumulated at the start of simulation for all defined components. The sparing cost is only charged at the time the spare is required by the system. At the same time, the ALDT cost is calculated by a per-unit of time factor multiplied by the user-defined ALDT time. These cost are calculated and the statistical parameters are updated during the repair of a failed block. The repair, whether handled by remove and replace or repair by adjustment, is subject to a per-unit of time cost as well. The time to repair is multiplied by this cost parameter to determine the repair cost. This figure is added into the cost calculation, and the overall value is reported to the user.

The output cost parameter is sent to a file during the DoEndOfRunUpdate in the Engine module. The maintainability cost is combined with the operational cost discussed in the previous section, and the total cost is reported along with the mean and standard

deviation over the specified number of runs. The block and node costs are reported separately to allow the user to analyze them individually. The case study provided following this section provides insight into how the user can use maintainability cost to evaluate system performance. The scenario uses maintainability cost to evaluate the trade-off between alternative maintenance strategies. As mentioned earlier, the flexibility to add multiple maintenance actions is an improvement to the current RAPTOR software. This case study integrates both advanced features and accounts for increased modeling accuracy.

CASE STUDY FOR MAINTAINABILITY COST

The case study assumes both maintenance actions are feasible within the given system. Components are fixed by remove and replace or repair by adjustment only. The cost for competing strategies is different, and the overall maintainability cost is the parameter for system evaluation. Remove and replace maintenance actions are quicker to perform, hence, cheaper in terms of labor. However, the spares required to support this strategy are expensive and are analyzed as a trade-off to repairing components through additional hours of manpower. The system is described by the reliability block diagram below and each component is defined with repair and sparing cost.

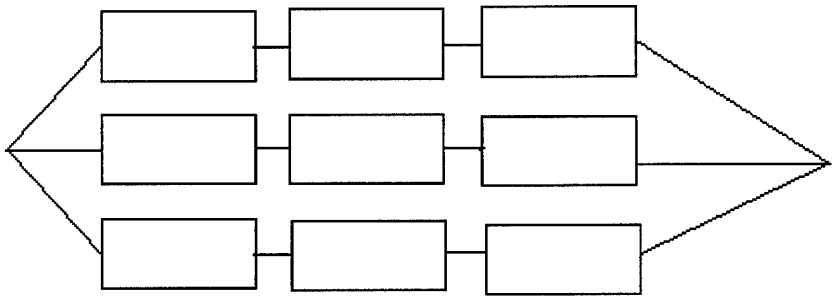


FIGURE 3-7. Reliability Block Diagram for Modeling Maintenance Costs

All blocks fail exponentially with mean 200 hours. The remove and replace distribution is lognormal with mean 10 hours and standard deviation 1 hour. The repair by adjustment distribution is lognormal with mean 100 hours and standard deviation 10 hours. The cost of remove and replace is $10 per unit time and repair by adjustment is $20 per unit time. The initial cost for all blocks is $1000 with a spare cost of $8000. The case study assumes an availability requirement of greater than 80%. There is no cost penalty for availability difference above 80%.

The simulation runs for 10,000 hours to determine the cost per hour of maintaining the system. The percent of maintenance actions handled by repair versus remove and replace is adjusted to determine an optimal value. The system begins with only 20% of the maintenance handled by repair and increase up to 50%. The percentage is incremented in 10% steps, and the overall cost figures, per hour, are reported. The cost values are $342, $340, $303, $321, while the availability decreases accordingly (96%,

94%, 93%, 88%). The system, therefore, performs best when 40% of the maintenance actions are handled by repair.

There are many additional factors that could influence this optimal percentage. Initially, the increment step would have to be decreased to obtain a more accurate answer. The optimal percentage may be slightly higher or lower than 40%. A cost factor could be applied to the output node to penalize the system based on availability cost. This would take advantage of the operational cost covered in the previous section and could change the optimal condition. Further manipulation of the labor cost, specifically the ratio between remove and replace and repair by adjust could influence the outcome of this scenario. All of these conditions would have to be defined by the user based on the physical systems and the criteria of the analysis. Overall, the maintainability cost structure provides additional insight into system performance provided by this enhancement.

ALGORITHM FOR CHANGES TO EMERGENCY SPARING

Step 1 – Define the added variable for ordering emergency spares (new code).

Step 2 – Determine if spares are ordered upon depletion or upon first request for missing spares (new code).

Step 3 – Determine the number of emergency spares to order (new code).

Step 4 – Replenish the emergency spare pool at the appropriate time and with the appropriate number of units (new code).

Step 5 – Allow the system to operate under normal conditions with the new sparing structure (RAPTOR).

EXPLANATION OF CHANGES TO EMERGENCY SPARING

The current version of RAPTOR allows the user to define emergency spare requirements when working with spare pools. The spares are ordered when a component fails, and no spares are available for repair. The user defines the amount of time required for spare arrival, thus effecting the length of component down time. Only one spare is ordered at the time of request, and the spare pool remains empty as the emergency spare is immediately used by the system. Emergency spares can not be ordered upon depletion of the spare pool as a preventive measure. This guarantees the system will experience additional delays if those components requiring this type of spare fail.

The changes to the RAPTOR code are aimed directly at giving the user additional flexibility in defining the given system. In many cases, the user would choose to order additional spares upon depletion of the spare pool. Also, the user may choose to order more than one spare to minimize the number of emergency spare acquisitions. This may be due to shipping cost or due to the time delay when waiting for the emergency spares to arrive. Both of these options are implemented in the code, and the user defines these parameters while creating the reliability block diagram.

The Dialog module allows the user to define the new parameters for the block definitions. The emergency spares can be ordered upon depletion of available spares in a spare pool or upon request. Ordering upon depletion takes place when the last spare is used from the inventory. This is a preventive measure, hoping the emergency spares arrive before they are required. Ordering upon request takes place when a component fails and there are no spares available. The user's option is stored as a Boolean variable

and affects the implementation of the remaining simulation code. The quantity ordered can be set to any integer value and is stored as an additional parameter. Prior to this change, only one emergency spare was ordered. This may directly affect shipping and logistic costs.

The FileN module is modified to handle the input and output of these new parameters to the storage files. The RBDwin and RBDBlock modules are also modified as the parameters are passed to and from the subroutines. Once the reliability block diagram is constructed, the simulation changes based on the user's selection. The Operate sub-routine orders the emergency spares for the spare pool. The spare pool decrements each time a repair takes place. The pool is replenished when this number drops to zero or when the next component fails. The software handles this using an "IF statement" to determine if the spares are ordered upon depletion (spare pool equal zero) or when the next component fails (upon request). The OrderASpare sub-routine is called, and the pool number is incremented accordingly. This code was changed by changing the increment number; the remainder of the sub-routine was untouched.

A detailed case study is included in Chapter Four as part of a research project with Motorola. The project examines a communications satellite network and the reliability of the physical system. The satellites form a constellation, and the failed units pull spares from a shared pool. Emergency spares are requested from ground stations, and the reliability is impacted based on the user's description of the ordering policy. The spare pool holds a varying number of spares. This scenario will be analyzed in greater detail and the benefits of these code changes will be highlighted. The significance of this

software enhancement is the added flexibility allowing the user to perform analysis that could not previously be accomplished.

Summary

The changes to the RAPTOR simulation code provide additional insight into reliability analysis. Many of the modifications are based on input from current users and shortcomings identified during software use. Other changes were prompted by ongoing research with local companies performing reliability analysis using RAPTOR. The analysis is supported by RAPTOR output, but minor modifications prove advantageous in further studies. The RAPTOR software was separately modified (in conjunction with this research) and the application specific software was turned over to those case scenario companies.

A brief review of the simulation changes highlights their purpose for inclusion in this research. The priority queueing modification is based on Operations Research theory verifying the importance of placing priorities on limited resources. The stream dependency enhancement mirrors the original concept of system dependency; however, the modeling is taken one step further. The physical system described by the reliability block diagram can cause components to be dependent on each other and/or the system status to be operational. The repair by adjustment changes to the maintenance capability of RAPTOR also stemmed from user input. System maintenance requirements are complex in nature and additional flexibility in this area is beneficial to the practicing engineer.

Reliability improvement and degradation was added to prevent all systems from being modeled as "good as new." The component's reliability may increase or decrease as a result of external and internal conditions as the system accumulates hours of operation. The block level modification and the changes to the spare pool using emergency sparing were directly required to assist local companies in ongoing reliability analysis. These changes also demonstrate the feasibility of modifying generic simulation code when the user has a higher level of simulation expertise. Finally, the cost structure, both the operations and maintainability costs, were included due to input from current users. The costs can be viewed as productivity at the same time.

The changes to the simulation code are highlighted with applicable case studies. These case studies are very short and simplified to demonstrate the new software capability, one modification at a time. Real-word applications would involve a greater level of detail and analysis. The intent of the case studies is to compare old and new versions of the reliability software and give the reader insight into the significance of the code changes. The case studies provide openings to new research and further studies for future work. Chapter Four demonstrates the utility of advancing reliability analysis using RAPTOR software. Three separate studies provide new research for the academic and the industrial community simultaneously. There is a great deal of potential for improving the current level reliability, maintainability and availability analysis in the engineering field.

APPENDIX 3A

DEFINITION FILES

```
        DIALOGS INFORMATION
        21 LINES REMOVED
  DistroBoxObj = OBJECT (HelpBoxObj);
    failureDistribution,
    failureParameters,
    repairDistribution,
    repairParameters,
    arepairDistribution,
    arepairParameters       : INTEGER;
    {repair by adjustment variables}
    probAdjustment,
    rradjFact,
    radjFact,
    rrlimFact,
    rlimFact,
    rrCost,
    rCost   : REAL;
    useAdjustment : BOOLEAN;
    {The parameters listed above include remove and replace factors for reliability improvement
    and degradation, also limiting factors.
    They also include cost factors for repair}
    ASK METHOD GetDistro    (INOUT fDistType, fNumParms, rDistType, rNumParms, arDistType, arNumParms : INTEGER;
                  INOUT fParmVals, rParmVals, arParmVals, rFactVals    : valueArray;
                  INOUT pAdjustment :  REAL;
                  INOUT uAdjustment :  BOOLEAN);
        13 LINES REMOVED
  PropertiesBoxObj = OBJECT (HelpBoxObj);
    ASK METHOD GetProperties(INOUT blockName              : STRING;
                  INOUT intArray              : integerArray;
                  INOUT failVal, repairVal, arepairVal, realArray : valueArray;
                  INOUT boolsArray            : boolArray;
                  INOUT poolName, resourceName      : STRING;
                  INOUT sparingType           : SparingType);
  END OBJECT; {PropertiesBoxObj}
  NodeBoxObj = OBJECT (HelpBoxObj);
    ASK METHOD GetNodeInput(INOUT typeNode                    : INTEGER;
                  IN numLinksOut, numLinksIn          : INTEGER;
                  INOUT cancelledFlag, isStartNode, isEndNode : BOOLEAN;
                  INOUT ngcost,nycost,nrcost          : REAL);
    OVERRIDE
        30 LINES REMOVED
END MODULE. {dmod Dialogs}
        ENGINE INFORMATION
        17 LINES REMOVED
BlockObj = OBJECT;
  BlockID       : INTEGER;
  BlockName     : STRING;
  Type          : BlockType;
  Gcost, Ycost, Rcost : REAL;
  NodeCost,
  BlockCost,
  RRCost,
  RCost,
  SpareCost,
  ALDTCost,
  {Cost factors for maintainability}
  LastNodeColorChange,
  LastBlockColorChange,
  NodeTimeToFail,
  BlockTimeToFail  : REAL;
  {Time tracking variables used to compute cost}
```

```
  NumPathsIn,
  GoodPathsRequired,
  SysStrm     : INTEGER;
  SystemDependence,  {TRUE if component stops accumulating time
                      when the system is down}
  UsesResources,
  EventBlock,
  UseAdjust    : BOOLEAN;
  FailSeedNumber,
  RepairSeedNumber,
  ARepairSeedNumber,
  StreamDependence          : INTEGER;
  InitNumberOfSpares        : INTEGER;
  DelayForSpares,
  ProbAdjust,
  UCost,
  DCost,
  FCost,
  SCost,
  ICost,
  LFactor,
  RRAdjfactor,
  RAdjfactor,
  RRLimfactor,
  RLimfactor       : REAL;
  {Cost factors and adjustment and limitation factors, provide a degradation/growth
  factor along with a limiting value.  Also provide cost estimates.}
  ReplenishQuantity :  INTEGER;
  SpareReplenishTime: REAL;
  NumDownStreamComp : INTEGER;
  FailDist,
  NumFailDistParam,
  RepairDist,
  NumRepairDistParam,
  ARepairDist,
  NumARepairDistParam : INTEGER;
  LogisticsDelay   : REAL;
  InfiniteSpares,
  RRFlag        : BOOLEAN;
  SparingMethod    : SparingType;
  PoolName,
  ResName        : STRING;
  PoolNumber,
  ResNumber,
  NumResRequired       : INTEGER;
  spcResPriority       : REAL;  {priority use of limited resources}
  FailDistParam,
  RepairParam,
  ARepairParam     : ParamArray;
  DownStreamBlock   : ARRAY INTEGER OF INTEGER;
  NumGoodPaths,
  NumActivePaths,
  DelayCounter,
  FailureCounter,
  RRFailureCounter,
  RFailureCounter  :INTEGER;
  {Counters used to determine reliability growth and degradation accordingly}
  Status,
          31 LINES REMOVED
 StatsReset       :BOOLEAN;
  FinalArray,FinCostArray       :FinalArrayType;
PROCEDURE StartEngine(IN StopTime, startStat, TimeSlice:REAL;IN NumRuns:INTEGER;
            IN NumberOfBlocks:INTEGER);
PROCEDURE GetSimOptions(IN c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c22, c23,c24 : BOOLEAN);
PROCEDURE DoEndOfRunUpdate(IN Availability, MTBDE, MDT, MTBMC, MRT, MRRT, MART,
                  GreenPercent, YellowPercent, RedPercent,TNodeCost, TBlockCost : REAL);
```

```
PROCEDURE DoEndOfSimUpdate(IN NumRuns : INTEGER;
                  IN AvailMon, MTBDEMon, MDTMon, MTBMCMon, MRTMon,MRRTMon, MARTMon, GreenPercentMon,
                    YellowPercentMon, RedPercentMon, TermMon, TNodeMon, TBlockMon : RStatObj);
PROCEDURE OpenOutFiles;
PROCEDURE CloseOutFiles;
END MODULE.
          FILEN INFORMATION
          12 LINES REMOVED
    repSeed,
    arepSeed,
    numSpares,
    failDist,
    numFailParams,
    repDist,
    numRepParams,
    arepDist,
    numARepParams,
    numDownstream,
          9 LINES REMOVED
    ALDTDelay,
    ALDTCostVal,
    SpareCostVal,
    numPerVal,
    pSuccess,
    uCostFactor,
    sCostFactor,
    dCostFactor,
    fCostFactor,
    iCostFactor,
    lFactor,
    probAdjust,
    RRAdjFact,
    RRCostFact,
    RCostFact,
    RAdjFact,
    RRLimFact,
    RLimFact,
    oldgcost,
    oldycost,
    oldrcost  : REAL;
    {cost factors added, along with growth/degradation factors and limitations}
    depends,
    infiniteSpares,
    useAdjust     : BOOLEAN;
{these arrays are all fixed in size because variable length arrays (such as strings) cannot}
{be written to disk}
    name          : FIXED ARRAY[1..20] OF CHAR;
    failParams,
    repParams,
    arepParams    : FIXED ARRAY[1..4] OF REAL;
{the fixed size prevents us from writing empirical data into the record because we don't know}
          33 LINES REMOVED
          RBD BLOCK INFORMATION
          36 LINES REMOVED
    typeNode,
    goodPaths : INTEGER;
    ngcost, nycost, nrcost : REAL;
    ASK METHOD SetType(IN nodeType : INTEGER);
    ASK METHOD SetGoodPaths(IN k   : INTEGER);
    ASK METHOD SetNum (IN label    : INTEGER);
    ASK METHOD SetKofN(IN k, n     : INTEGER);
    ASK METHOD SetNodeCost(IN costg, costy, costr : REAL);
    {Node cost method added to reset cost when statistics are started and program
     is initialized}
  END OBJECT; {RBDNodeObj}
  LinkObj = OBJECT(PolylineObj);
```

```
        17 LINES REMOVED
RBDBlockObj = OBJECT(RBDBasicObj);
  blockName,
  poolName,
  resourceName    : STRING;
  failDistro,
  numFailParams,
  failStream,
  repairDistro,
  numRepairParams,
  repairStream,
  arepairDistro,
  numARepairParams,
  arepairStream,
  numSpares,
  numResources,
  sparePoolNum,
  resPoolNum,
  streamDepend       : INTEGER;
  sparingType    : SparingType;
  spareRate,
  spareDelay,
  ALDTDelay,
  ALDTCostVal,
  SpareCostVal,
  numPerVal,
  probAdj,
  pSuccess,
  uCostFactor,
  sCostFactor,
  dCostFactor,
  fCostFactor,
  iCostFactor,
  lFactor,
  rradjfactor,
  rrcost,
  rcost,
  radjfactor,
  rrlimfactor,
  rlimfactor     : REAL;
  {Cost factors along with growth/degradation factors and limitations, also includes
  remove and replace versus repair variables}
  isConnectedNode,
  systemDepend,
  operatingBlock,
  useResource,
  useAdj,
  infiniteSpares  :BOOLEAN;
  failVals,
  repairVals,
  arepairVals    : valueArray;
  ASK METHOD SetBlockData   (IN bName                          : STRING;
                    IN fDist, numFParms, fStream, rDist, numRParms,
                      rStream, arDist, numARParms, arStream      : INTEGER;
                    IN fVals, rVals, arVals, rfactvals          : valueArray;
                    IN holdvals :  valueArray;
                    IN uAdj  :  BOOLEAN);
  ASK METHOD SetSpecialData  (IN realArray        : valueArray;
                    IN boolsArray        : boolArray;
                    IN bPoolName, bResName : STRING;
                    IN bNumRes,streamdepend   : INTEGER;
                    IN bSpareType        : SparingType);
  ASK METHOD DisplayDetails;
        11 LINES REMOVED
        RBD RES INFORMATION
        11 LINES REMOVED
```

```
  newSparesArrival,
  emergencyTime, emergencyNumber    : REAL;
  emergencyFlag    : BOOLEAN;
  ASK METHOD SetData(IN name                : STRING;
              IN type               : SparingType;
              IN poolNum, numSpares      : INTEGER;
              IN numSpareArrival, emerTime, emerNumber   : REAL;
              IN emerFlag : BOOLEAN);
  ASK METHOD RemovePool(IN name : STRING;
              IN type : SparingType);
        12 LINES REMOVED
        RBD WIN INFORMATION
        54 LINES REMOVED
PROCEDURE ClearAllBlocks;
PROCEDURE GetBlockName   (INOUT nameString                          : STRING;
              INOUT cancelledFlag, remain, sysDepen, operating   : BOOLEAN;
              INOUT failStreamNum, repairStreamNum, arepairStreamNum,
                 strmDepen                        : INTEGER;
              INOUT pS, uCost, sCost, dCost, fCost, iCost, lfact : REAL);
PROCEDURE InitGetDistro  (INOUT blockName                           : STRING;
              INOUT fDistroType, fNumParms, rDistroType, rNumParms,
                 arDistroType, arNumParms              : INTEGER;
              INOUT fParmVals, rParmVals, arParmVals, rFactorVals     : valueArray;
              INOUT prAdjustment : REAL;
              INOUT usAdjustment : BOOLEAN);
PROCEDURE GetStream    (IN   barLbl                          : STRING;
              INOUT strmNum                       : INTEGER);
PROCEDURE GetSpecial   (IN   barLbl                          : STRING;
              INOUT sparesVal, sparesReplenVal, sparesDelayVal,
                 ALDTVal,ALDTcost, Sparecost, numPerVal      : REAL;
              INOUT infinSpares, resources              : BOOLEAN;
              INOUT poolName, resName                   : STRING;
              INOUT numRes                        : INTEGER;
              INOUT spareType                     : SparingType);
PROCEDURE GetNodeType   (INOUT typeNode                          : INTEGER;
              IN   numLinksOut, numLinksIn           : INTEGER;
              INOUT cancelledFlag                 : BOOLEAN;
              INOUT ngcost,nycost,nrcost             : REAL);
PROCEDURE ClearObject;
        73 LINES REMOVED
```

APPENDIX 3B

IMPLEMENTATION FILES

```
        DIALOG MODULE INFORMATION
        45 LINES REMOVED
{holds return value of SendAlert call}
  grncost, ylwcost, redcost  : REAL;
        PROCEDURE GET ERROR BOX REMOVED
        PROCEDURE CONVERT TO STRING REMOVED
        PROCEDURE CALL HELP REMOVED
        PROCEDURE SEND ALERT REMOVED
        PROCEDURE GET PARAMETERS REMOVED
        ENTIRE HELP BOX OBJ REMOVED
        ENTIRE ZOOM BOX OBJ REMOVED
        DISTRO BOX OBJ INFORMATION
OBJECT DistroBoxObj;
        GET DISTRO INFORMATION
  ASK METHOD GetDistro(INOUT fDistType, fNumParms, rDistType, rNumParms,
                arDistType, arNumParms : INTEGER;
              INOUT fParmVals, rParmVals, arParmVals, rFactVals     : valueArray;
              INOUT pAdjustment : REAL;
              INOUT uAdjustment : BOOLEAN);
  VAR
    i       : INTEGER;
    fFlag,
    rFlag,
    arFlag,
    fEmpFlag,
    rEmpFlag,
    arEmpFlag   : BOOLEAN;              {Flags to determine if I got good values and can continue}
    distroName  : STRING;
    failCombo,
    repairCombo,
    arepairCombo: ComboBoxObj;            {Fail and repair combo boxes}
    fTags,
    rTags,
    arTags      : stringArray;          {Array of strings holding parameter names}
    fLabel,
    rLabel,
    arLabel     : ARRAY INTEGER OF LabelObj;   {Labels that hold parameter names in DB}
    fVals,
    rVals,
    arVals      : ARRAY INTEGER OF ValueBoxObj; {Values that hold parameters in DB}
    probVal,
    rradjustFact,
    radjustFact,
    rrlimitFact,
    rlimitFact,
    {Reliability growth/degradation factors and their limitations for both remove and replace,
    and repair maintenance options.}
    rrcostVal,
    rcostVal    : ValueBoxObj;
    {Cost values for remove and replace and repair maintenance options.}
    useVal      : CheckBoxObj;
  BEGIN
    NEW(fLabel,1..4);
    NEW(rLabel,1..4);
    NEW(arLabel,1..4);
    NEW(fVals, 1..4);
    NEW(rVals, 1..4);
    NEW(arVals,1..4);
    NEW(fTags, 1..4);
    NEW(rTags, 1..4);
    NEW(arTags,1..4);
```

```
failureDistribution := fDistType;          {Left-side variables are internal to this DB}
failureParameters  := fNumParms;           {check Ddialogs.mod}
repairDistribution  := rDistType;
repairParameters   := rNumParms;
arepairDistribution := arDistType;
arepairParameters  := arNumParms;
probAdjustment     := pAdjustment;
useAdjustment      := uAdjustment;
rradjFact          := rFactVals[1];
rrlimFact          := rFactVals[2];
radjFact           := rFactVals[3];
rlimFact           := rFactVals[4];
rrCost             := rFactVals[5];
rCost              := rFactVals[6];
failCombo   := Child("FailCombo", 931);     {Get hold of DB controls}
repairCombo  := Child("RepairCombo", 932);
arepairCombo := Child("ARepairCombo", 933);
probVal     := Child("AdjProb", 935);
useVal      := Child("adjsep", 934);
rradjustFact  := Child("RRAdjustFactor", 950);
rrlimitFact   := Child("RRLimitFactor", 954);
radjustFact   := Child("RAdjustFactor", 960);
rlimitFact    := Child("RLimitFactor", 964);
rrcostVal     := Child("RRCostBox", 970);
rcostVal      := Child("RCostBox", 971);
FOR i := 1 TO 4
  fVals[i]  := Child("FailVal", i);
  fLabel[i] := Child("FailLabel", i);
  rVals[i]  := Child("RepairVal", i);
  rLabel[i] := Child("RepairLabel", i);
  arVals[i] := Child("ARepairVal", i);
  arLabel[i] := Child("ARepairLabel", i);
END FOR;
ConvertToString(fDistType, distroName);     {Get the name of the failure distributon #}
ASK failCombo TO DisplayText(distroName);   {Ask combo box to display that name}
ConvertToString(rDistType, distroName);     {Get the name of the repair distributon #}
ASK repairCombo TO DisplayText(distroName); {Ask combo box to display that name}
ConvertToString(arDistType, distroName);
ASK arepairCombo TO DisplayText(distroName);
GetParameters(fDistType, fTags);            {Get failure parameter names}
GetParameters(rDistType, rTags);            {Get repair parameter names}
GetParameters(arDistType, arTags);
ASK probVal TO DisplayValue(pAdjustment);
ASK useVal TO DisplayCheck(uAdjustment);
ASK rradjustFact TO DisplayValue(rFactVals[1]);
ASK rrlimitFact TO DisplayValue(rFactVals[2]);
ASK radjustFact TO DisplayValue(rFactVals[3]);
ASK rlimitFact TO DisplayValue(rFactVals[4]);
ASK rrcostVal TO DisplayValue(rFactVals[5]);
ASK rcostVal TO DisplayValue(rFactVals[6]);
FOR i := 1 TO 4
       8 LINES REMOVED
  IF arTags[i] = ""
    ASK arVals[i] TO Deactivate;
  END IF;
END FOR;
Update;
       10 LINES REMOVED
IF arDistType <> 16
  FOR i := 1 TO arNumParms
    ASK arVals[i] TO DisplayValue(arParmVals[i]);
  END FOR;
END IF;
REPEAT                                {Repeat this loop until flags are valid}
  button := AcceptInput();            {Accept user input}
  fDistType := failureDistribution;   {Right-side variables get set thru BeSelected}
```

```
    fNumParms := failureParameters;        {method below so they are always current}
    rDistType := repairDistribution;
    rNumParms := repairParameters;
    arDistType := arepairDistribution;
    arNumParms := arepairParameters;
    pAdjustment := ASK probVal Value();
    uAdjustment := useAdjustment;
    rFactVals[1]   := ASK rradjustFact Value();
    rFactVals[2]   := ASK rrlimitFact Value();
    rFactVals[3]   := ASK radjustFact Value();
    rFactVals[4]   := ASK rlimitFact Value();
    rFactVals[5]   := ASK rrcostVal Value();
    rFactVals[6]   := ASK rcostVal Value();
    {These variables are stored in an array to provide access to and from procedure
    and method calls.}
        24 LINES REMOVED
    {This section handles the remove and replace options for maintenance.}
    IF arDistType = 16                {Same for repair values and distribution}
      arFlag := FALSE;
      GetEmpirical(arNumParms, arParmVals, arEmpFlag);
    ELSE
      arEmpFlag := FALSE;
      DISPOSE(arParmVals);
      NEW(arParmVals, 1..arNumParms);
      FOR i := 1 TO arNumParms
        arParmVals[i] := ASK arVals[i] Value();
      END FOR;
      CheckValidInput(arDistType, arNumParms, arParmVals, arFlag);
    END IF;
    {This section handles the repair options for maintenance.}
{Stay in this loop until good failure and repair flags}
  UNTIL ((fFlag OR fEmpFlag) AND (rFlag OR rEmpFlag) AND (arFlag OR arEmpFlag));
  DISPOSE(fLabel);
  DISPOSE(rLabel);
  DISPOSE(arLabel);
  DISPOSE(fVals);
  DISPOSE(rVals);
  DISPOSE(arVals);
  DISPOSE(fTags);
  DISPOSE(rTags);
  DISPOSE(arTags);
END METHOD;{GetDistro}
        BE SELECTED INFORMATION
        9 LINES REMOVED
  arepairCombo: ComboBoxObj;
  tags     : stringArray;          {Array of strings holding parameter names}
  label    : ARRAY INTEGER OF LabelObj;
  vals     : ARRAY INTEGER OF ValueBoxObj;
  probVal, rradjustFact, rrlimitFact, radjustFact, rlimitFact,
  rrcostVal, rcostVal  : ValueBoxObj;
  useVal   : CheckBoxObj;
BEGIN
        5 LINES REMOVED
  arepairCombo:= Child("ARepairCombo", 933);
  probVal   := Child("AdjProb", 935);
  useVal    := Child("adjsep", 934);
  rradjustFact := Child("RRAdjustFactor", 950);
  rrlimitFact  := Child("RRLimitFactor", 954);
  radjustFact  := Child("RAdjustFactor", 960);
  rlimitFact   := Child("RLimitFactor", 964);
  rrcostVal    := Child("RRCostBox", 970);
  rcostVal     := Child("RCostBox",971);
        16 LINES REMOVED
    ELSE
      isFailure := 3;
      distro := arepairCombo.Text();
```

```
        FOR i := 1 TO 4
          vals[i] := Child("ARepairVal", i);
          label[i] := Child("ARepairLabel", i);
        END FOR;
      END IF;
          7 LINES REMOVED
      ELSE
        arepairDistribution := distroNum;
        arepairParameters  := numParams;
      END IF;
          25 LINES REMOVED
    ELSIF lastClicked.Id = 934
      useAdjustment := useVal.Checked;
    ELSIF lastClicked.Id = 935
      probVal := Child("AdjProb", 935);
      {This section handles the use of repair and the probability of having a repair
       item versus a true remove and replace item.}
    ELSIF lastClicked.Id = 950
      rradjustFact := Child("RRAdjustFactor", 950);
    ELSIF lastClicked.Id = 954
      rrlimitFact  := Child("RRLimitFactor", 954);
    ELSIF lastClicked.Id = 960
      radjustFact  := Child("RAdjustFactor", 960);
    ELSIF lastClicked.Id = 964
      rlimitFact   := Child("RLimitFactor", 964);
      {This section sets the reliability growth/degradation factors and their limitations
       input from the user's dialog box.}
    ELSIF lastClicked.Id = 970
      rrcostVal := Child("RRCostBox", 970);
    ELSIF lastClicked.Id = 971
      rcostVal := Child("RCostBox",971);
      {This handles the remove and replace and repair cost parameters.}
    ELSIF lastClicked.Id = 945          {If 'Help' was clicked, call help}
     CallHelp("945");
    END IF;
    DISPOSE(label);
    DISPOSE(vals);
    DISPOSE(tags);
  END METHOD;{BeSelected}
          METHOD DETERMINE DISTRIBUTION REMOVED
          METHOD GET EMPIRICAL REMOVED
  END OBJECT; {DistroBoxObj}
          ENTIRE STREAM BOX OBJ REMOVED
          PROPERTIES BOX OBJ INFORMATION
OBJECT PropertiesBoxObj;
          GET PROPERTIES INFORMATION
  ASK METHOD GetProperties(INOUT blockName              : STRING;
                INOUT intArray              : integerArray;
                INOUT failVal, repairVal, arepairVal, realArray : valueArray;
                INOUT boolsArray            : boolArray;
                INOUT poolName, resourceName     : STRING;
                INOUT sparingType           : SparingType);
  VAR
{local variables}
    i,sdependName,strmDep          : INTEGER;
{a counter}
          20 LINES REMOVED
    useAdj       : BOOLEAN;
{tells whether to deactivate the 'accept remaining defaults' button}
    check1, check2,
    check3, check4 : CheckBoxObj;
{generic check boxes}
          10 LINES REMOVED
    arepairDistro,
    numARepairParams,
    arepairStream,
```

```
        7 LINES REMOVED
    upCost,
    standCost,
    level,
    downCost,
    failCost,
    initialCost,
    probAdj,
    rradjfactor,
    rrlimfactor,
    radjfactor,
    rlimfactor,
    ALDTCostVal,
    SpareCostVal,
    rrcost,
    rcost       : REAL;
    {The above parameters were added to handled added features of cost, reliability growth/
    degradation, maintenance by repair versus remove and replace.}
    holdArray    : valueArray;
  BEGIN
{get handles to all the check boxes and labels in this dialog box}
        16 LINES REMOVED
    useAdj        := boolsArray[5];
        7 LINES REMOVED
    arepairDistro  := intArray[8];
    numARepairParams := intArray[9];
    arepairStream   := intArray[10];
        7 LINES REMOVED
    probAdj       := realArray[7];
    rradjfactor     := realArray[9];
    rrlimfactor     := realArray[10];
    radjfactor      := realArray[11];
    rlimfactor      := realArray[12];
    upCost        := realArray[13];
    downCost      := realArray[14];
    failCost      := realArray[15];
    rrcost        := realArray[16];
    rcost         := realArray[17];
    ALDTCostVal    := realArray[18];
    SpareCostVal   := realArray[19];
    initialCost    := realArray[20];
    standCost      := realArray[21];
    level         := realArray[22];
    tempName       := blockName;
  IF NOT operatingBlock  {If an event block, disable the accept defaults button and call GetBlockName}
    remaining := TRUE;
    GetBlockName(blockName, xcelFlag, remaining, sysDep, operatingBlock,
            failStream, repairStream, arepairStream,strmDep, probSuccess,
            upCost, standCost, downCost, failCost, initialCost, level);
        7 LINES REMOVED
    DISPOSE(arepairVal);
        7 LINES REMOVED
      arepairDistro := 7;
      numARepairParams := 2;
      NEW (repairVal, 1..numRepairParams);
      repairVal[1] := 1.;  {set lognormal parameter values}
      repairVal[2] := 1.;
      NEW (arepairVal, 1..numARepairParams);
      arepairVal[1] := 1.;
      arepairVal[2] := 1.;
      probAdj := 0.0;
      upCost := 0.0;
      downCost := 0.0;
      standCost := 0.0;
      level     := 1.0;
      failCost := 0.0;
```

```
      initialCost := 0.0;
      useAdj := FALSE;
      rradjfactor := 1.0;
      rrlimfactor := 2.0;
      radjfactor := 1.0;
      rlimfactor := 2.0;
      rrcost  := 0.0;
      rcost  := 0.0;
      infiniteSpares := TRUE;  {default to infinite spares}
      sparingType := Infinite;
{get distribution info}
      NEW(holdArray, 1..6);
      holdArray[1] := rradjfactor;
      holdArray[2] := rrlimfactor;
      holdArray[3] := radjfactor;
      holdArray[4] := rlimfactor;
      holdArray[5] := rrcost;
      holdArray[6] := rcost;
    {Variables put into temporary array to be sent into procedure.}
      InitGetDistro(blockName, failDistro, numFailParams, repairDistro,
              numRepairParams, arepairDistro, numARepairParams,
              failVal, repairVal, arepairVal, holdArray, probAdj,
              useAdj);
      rradjfactor := holdArray[1];
      rrlimfactor := holdArray[2];
      radjfactor  := holdArray[3];
      rlimfactor  := holdArray[4];
      rrcost      := holdArray[5];
      rcost       := holdArray[6];
      DISPOSE(holdArray);
{get special repair info}
      GetSpecial(blockName + "'s Maintenance Information", nSpares, spareRep, spareDel,
            ALDTDelay, ALDTCostVal, SpareCostVal, numPerVal, infiniteSpares,
            useResource, poolName, resourceName, numResources, sparingType);
    END IF;
          63 LINES REMOVED
        remaining := TRUE;
        GetBlockName(blockName, xcelFlag, remaining, sysDep, operatingBlock,
              failStream, repairStream, arepairStream,strmDep, probSuccess,
              upCost, standCost, downCost, failCost, initialCost, level);
          11 LINES REMOVED
      NEW(holdArray, 1..6);
      holdArray[1] := rradjfactor;
      holdArray[2] := rrlimfactor;
      holdArray[3] := radjfactor;
      holdArray[4] := rlimfactor;
      holdArray[5] := rrcost;
      holdArray[6] := rcost;
        InitGetDistro(blockName, failDistro, numFailParams, repairDistro,
              numRepairParams, arepairDistro, numARepairParams,
              failVal, repairVal, arepairVal, holdArray, probAdj,
              useAdj);
      rradjfactor := holdArray[1];
      rrlimfactor := holdArray[2];
      radjfactor  := holdArray[3];
      rlimfactor  := holdArray[4];
      rrcost      := holdArray[5];
      rcost       := holdArray[6];
      DISPOSE(holdArray);
{get new failure distribution}
      END IF;
      IF specialFlag AND operatingBlock
{if user clicked last check box}
        GetSpecial(blockName + "'s Maintenance Information", nSpares, spareRep, spareDel,
              ALDTDelay, ALDTCostVal, SpareCostVal, numPerVal, infiniteSpares,
              useResource, poolName, resourceName, numResources, sparingType);
```

```
{get new special repair info}
      END IF;
    END IF;
  END IF;
{Put all the new values back into the array to be passed back to the calling function}
    boolsArray[1] := sysDep;
    boolsArray[2] := infiniteSpares;
    boolsArray[3] := useResource;
    boolsArray[4] := operatingBlock;
    boolsArray[5] := useAdj;
          7 LINES REMOVED
    intArray[8] := arepairDistro;
    intArray[9] := numARepairParams;
    intArray[10] := arepairStream;
    intArray[11] := strmDep;
    realArray[1] := nSpares;
    realArray[2] := spareRep;
    realArray[3] := spareDel;
    realArray[4] := ALDTDelay;
    realArray[5] := numPerVal;
    realArray[6] := probSuccess;
    realArray[7] := probAdj;
    realArray[9] := rradjfactor;
    realArray[10] := rrlimfactor;
    realArray[11] := radjfactor;
    realArray[12] := rlimfactor;
    realArray[13] := upCost;
    realArray[14] := downCost;
    realArray[15] := failCost;
    realArray[16] := rrcost;
    realArray[17] := rcost;
    realArray[18] := ALDTCostVal;
    realArray[19] := SpareCostVal;
    realArray[20] := initialCost;
    realArray[21] := standCost;
    realArray[22] := level;
  END METHOD; {GetProperties}
END OBJECT; {PropertiesBoxObj}
          NODE BOX OBJ INFORMATION
OBJECT NodeBoxObj;
          GET NODE INPUT INFORMATION
  ASK METHOD GetNodeInput(INOUT typeNode                    : INTEGER;
                IN numLinksOut, numLinksIn          : INTEGER;
                INOUT cancelledFlag, isStartNode, isEndNode : BOOLEAN;
                INOUT ngcost,nycost,nrcost          : REAL);
  VAR
{local variables}
    radioButton : RadioButtonObj;
{a generic radio button}
    gcostBox, ycostBox, rcostBox : ValueBoxObj;
          15 LINES REMOVED
    gcostBox := Child("GreenCost",814);
    ycostBox := Child("YellowCost", 815);
    rcostBox := Child("RedCost", 816);
    ASK gcostBox TO DisplayValue(ngcost);
    ASK ycostBox TO DisplayValue(nycost);
    ASK rcostBox TO DisplayValue(nrcost);
    button := AcceptInput();
    ngcost := ASK gcostBox Value();
    nycost := ASK ycostBox Value();
    nrcost := ASK rcostBox Value();
    {Determines the user input values for the node cost, green, yellow and red.}
    IF ASK button ReferenceName = "OKButton"
          28 LINES REMOVED
  END METHOD; {GetNodeInput}
```

```
            BE SELECTED INFORMATION
  ASK METHOD BeSelected;
  VAR
   gcostBox, ycostBox, rcostBox : ValueBoxObj;
  BEGIN
            19 LINES REMOVED
WHEN 814:
        gcostBox := Child("GreenCost",814);
      WHEN 815:
        ycostBox := Child("YellowCost", 815);
      WHEN 816:
        rcostBox := Child("RedCost", 816);
      {Determines the user input node cost, green, yellow and red.}
      OTHERWISE
    END CASE;
  END METHOD; {BeSelected}
END OBJECT; {NodeBoxObj}
            SPECIAL BOX INORMATION
OBJECT SpecialBoxObj;
            BE SELECTED INFORMATION
  ASK METHOD BeSelected;
  VAR
    value1, value2,
    value3, value4,
    value5, value6, value7, value8  : ValueBoxObj;
    label1, label2,
    label3        : LabelObj;
{generic value boxes}
            30 LINES REMOVED
  value7 :=    Child("ALDTCostBox", 851);
  value8 :=    Child("SpareCostBox", 850);
    lastClicked := ASK SELF LastPicked;
            170 LINES REMOVED
    END CASE;
  END METHOD; {BeSelected}
END OBJECT; {SpecialBoxObj}
            ENTIRE SIM BOX OBJ REMOVED
            ENTIRE STATUS BOX OBJ REMOVED
            ENTIRE ABOUT BOX OBJ REMOVE
            TABLE BOX INFORMATION
OBJECT TableBoxObj;
            BE SELECTED INFORMATION
  ASK METHOD BeSelected;
  VAR
    validFails,
    validRepairs,
    validARepairs : BOOLEAN;
            8 LINES REMOVED
    newARvals, newholdArray, newholdArray2   : valueArray;
  BEGIN
            66 LINES REMOVED
        NEW(newARvals, 1..block.numARepairParams);
        validFails   := TRUE;
        validRepairs := TRUE;
        validARepairs := TRUE;
            14 LINES REMOVED
           NEW(newholdArray, 1..4);
            newholdArray[1] := block.rradjfactor;
            newholdArray[2] := block.rrlimfactor;
            newholdArray[3] := block.radjfactor;
            newholdArray[4] := block.rlimfactor;
            newholdArray[5] := block.rrcost;
            newholdArray[6] := block.rcost;
            NEW(newholdArray2, 1..4);
            newholdArray2[1] := block.probAdj;
            newholdArray2[2] := block.uCostFactor;
```

```
            newholdArray2[3] := block.dCostFactor;
            newholdArray2[4] := block.fCostFactor;
            newholdArray2[5] := block.sCostFactor;
            newholdArray2[6] := block.lFactor;
            ASK block TO SetBlockData(block.blockName, block.failDistro, block.numFailParams,
                        block.failStream, block.repairDistro,
                        block.numRepairParams, block.repairStream,
                        block.arepairDistro, block.numARepairParams,
                        block.arepairStream, newFvals, newRvals, newARvals,
                        newholdArray, newholdArray2, block.useAdj);
            DISPOSE(newholdArray);
            DISPOSE(newholdArray2);
            somethingChanged := TRUE;
          END IF;
        END IF;
            14 LINES REMOVED
          NEW(newholdArray, 1..4);
            newholdArray[1] := block.rradjfactor;
            newholdArray[2] := block.rrlimfactor;
            newholdArray[3] := block.radjfactor;
            newholdArray[4] := block.rlimfactor;
            newholdArray[5] := block.rrcost;
            newholdArray[6] := block.rcost;
            NEW(newholdArray2, 1..6);
            newholdArray2[1] := block.probAdj;
            newholdArray2[2] := block.uCostFactor;
            newholdArray2[3] := block.dCostFactor;
            newholdArray2[4] := block.fCostFactor;
            newholdArray2[5] := block.sCostFactor;
            newholdArray2[6] := block.lFactor;
            ASK block TO SetBlockData(block.blockName, block.failDistro, block.numFailParams,
                        block.failStream, block.repairDistro,
                        block.numRepairParams, block.repairStream,
                        block.arepairDistro, block.numARepairParams,
                        block.arepairStream, newFvals, newRvals, newARvals,
                        newholdArray, newholdArray2,block.useAdj);
            DISPOSE(newholdArray);
            DISPOSE(newholdArray2);
            somethingChanged := TRUE;
            13 LINES REMOVED
  END METHOD; {BeSelected}
            METHOD OBJ TERMINATE REMOVED
            METHOD SET LABELS REMOVED
            METHOD MANAGE LABELS REMOVED
            METHOD HIGHLIGHT REMOVED
            METHOD SORT BLOCKS REMOVED
            METHOD INSERT BLOCK REMOVED
            METHOD PRINT REMOVED
            DETAILS BOX OBJ INFORMATION
OBJECT DetailsBoxObj;
            BE SELECTED INFORMATION
  ASK METHOD BeSelected;
  VAR
    rVal, uVal, dVal, fVal, iVal    : ValueBoxObj;
    remains : ButtonObj;
  BEGIN
    rVal := Child("RValBox", 920);
    uVal := Child("UCostBox", 923);
    dVal := Child("DCostBox", 922);
    fVal := Child("FCostBox", 921);
    iVal := Child("ICostBox", 924);
    remains := Child("RemainButton", 915);
    lastClicked := ASK SELF LastPicked;   {Find what was last clicked on}
    IF lastClicked.Id = 919  {If block type radio box, find which button}
      lastSelected := ASK lastClicked LastPicked;
      IF lastSelected.Id = 9191 {Operating}
```

```
        ASK rVal TO Deactivate;
        ASK fVal TO Deactivate;
        ASK uVal TO Activate;
        ASK dVal TO Activate;
        IF NOT noRemain
          ASK remains TO Activate;
        END IF;
      ELSE                {Event}
        ASK rVal   TO Activate;
        ASK remains TO Deactivate;
        ASK fVal TO Activate;
        ASK uVal TO Deactivate;
        ASK dVal TO Deactivate;
      END IF;
    END IF;
    INHERITED BeSelected;
  END METHOD; {BeSelected}
END OBJECT;
          POOLS BOX OBJ INFORMATION
OBJECT PoolsBoxObj;
          BE SELECTED INFORMATION
  ASK METHOD BeSelected;
  VAR
          13 LINES REMOVED
    emergencyTime, emergencyNumber   : REAL;
    value1, value2,
    value3, value4, value5   : ValueBoxObj;
    check1, check2, check3   : CheckBoxObj;
    combo          : ComboBoxObj;
    poolRadBox      : RadioBoxObj;
    label1, label2,
    label3, label4,
    label5, label6, label7, label8          : LabelObj;
          12 LINES REMOVED
    value5 := Child("Value5" ,568);
          30 LINES REMOVED
        ASK value4 TO SetHidden(FALSE);
        ASK value5 TO SetHidden(FALSE);
          18 LINES REMOVED
        ASK value4 TO SetHidden(TRUE);
        ASK value5 TO SetHidden(TRUE);
          18 LINES REMOVED
        ASK value4 TO SetHidden(TRUE);
        ASK value5 TO SetHidden(TRUE);
          26 LINES REMOVED
        ASK value4 TO Activate;
        ASK value5 TO Activate;
        ASK label4 TO Activate;
        ASK label5 TO Activate;
          5 LINES REMOVED
        ASK value4 TO Deactivate;
        ASK value5 TO Deactivate;
        ASK label4 TO Deactivate;
        ASK label5 TO Deactivate;
          15 LINES REMOVED
          ASK value4 TO DisplayValue(pool.emergencyTime);
          ASK value5 TO DisplayValue(pool.emergencyNumber);
          23 LINES REMOVED
        ASK value4 TO Activate;
        ASK value5 TO Activate;
        ASK label4 TO Activate;
        ASK label5 TO Activate;
          5 LINES REMOVED
        ASK value4 TO Deactivate;
        ASK value5 TO Deactivate;
```

```
          ASK label4 TO Deactivate;
          ASK label5 TO Deactivate;
            44 LINES REMOVED
            emergencyTime := value4.Value();
            emergencyNumber := value5.Value();
            emergencyFlag   := check3.Checked;
          END IF;
        ELSE
          emergencyTime := 0.0;
          emergencyNumber := 0.0;
          emergencyFlag := FALSE;
        END IF;
            55 LINES REMOVED
              ASK poolGroup TO RemoveThis(pool);
              NEW(pool);
              ASK pool TO SetData(name, type, poolNum, newSpares,
                        newSparesArrival, emergencyTime, emergencyNumber,
                         emergencyFlag);
              somethingChanged := TRUE;
              EXIT;
            END IF;
          END IF;
        END FOREACH;
      END IF;
    END IF;
{If I'm doing OK AND I'm not updating, enter here}
    IF ((validName) AND (goodValue) AND (NOT update))
      NEW(pool);
      ASK pool TO SetData(name, type, poolNum, newSpares,
                newSparesArrival, emergencyTime, emergencyNumber, emergencyFlag);
      somethingChanged := TRUE;
            70 LINES REMOVED
        ASK value4 TO DisplayValue(0.);
        ASK value5 TO DisplayValue(0.);
        ASK value2 TO Deactivate;
        ASK value3 TO Deactivate;
        ASK value4 TO Deactivate;
        ASK value5 TO Deactivate;
            52 LINES REMOVED
END OBJECT; {PoolsBoxObj}
          ENTIRE SORTING BOX OBJ REMOVED
END MODULE. {imod Dialogs}
          ENGINE INFORMATION
IMPLEMENTATION MODULE engine;
          23 LINES REMOVED
VAR
  intArray        : integerArray;
  NumberOfBlocks,
  counta, countb, countc, countd, counte, countg, countf:INTEGER;
          15 LINES REMOVED
  CompTimeToARepair,
  GreenTime,
  YellowTime,
  RedTime,
  TotalBlockCost,
  TotalNodeCost     :LMONITORED REAL BY RStatObj;
  {these variables are used within a run and are reset at the end of each run}
  LastColorChange,
  LastUpDownChange,
  temp4           :REAL;
  {these are used to determine the duration of green/yellow/red events and up/down events}
  SparesTrackingGroup,
  {group containing a list of components which do not have infinite spares}
  SysDepGroup, Strm1DepGroup,
  Strm2DepGroup, Strm3DepGroup  :QueueObj;
  {group containing a list of components which are system dependent}
```

```
GroupMember,Group1Member,
Group2Member, Group3Member  :BlockObj;
{a generic BlockObj}
SystemTrigger, Stream1Trigger,
Stream2Trigger, Stream3Trigger  :TriggerObj;
{the trigger that system dependent components are told to wait for when system is down}
{also include the stream dependent triggers which components are told to wait for when
a particlur stream is down.}
        27 LINES REMOVED
CompRRepairMon,
CompARepairMon,
GreenMon,
YellowMon,
RedMon, TNodeMon, TBlockMon     :RStatObj;
CustSpareMon,
PoolMon         :ARRAY INTEGER OF ITimedStatObj;
{these stat monitors provide statistics on the "within a run" variables of interest}
TimeTerminated,GraphicsOutput,Aborted,KeepSimStats,KeepRunStats,OutOfSpares,
StatsStarted, AvailFile,MTBDEFile,MDTFile,MTBMCFile,MRTFile,SysFailTimeFile,
SysRepTimeFile,EventsFile, SparesFile, EORFile, FinalFile,MRRTFile,MARTFile,
TieBreaking, EndSimTimeFile, TestMode, ShowSimProgress,ShowEORstats  : BOOLEAN;
{boolean variables used to define simulation options}
StopCriteria,
        7 LINES REMOVED
ARepairStream   : ARRAY INTEGER OF RandomGenObj;
{random number streams}
i, p        : INTEGER;
{indexing variables}
dialogBox     : DialogBoxObj;
EventController : RBDSimControlObj;
{this is used to break ties when several events are scheduled at the same time}
AoStream,
NCStream,
BCStream,
LStream,
MTBDEStream,
MDTStream,
MTBMCStream,
MRTStream,
MRRTStream,
MARTStream,
FailTimeStream,
RepTimeStream,
ARepTimeStream,
JRepTimeStream,
EventStream,
        8 LINES REMOVED
        START ENGINE INFORMATION
PROCEDURE StartEngine(IN simLength, startStat, TimeSlice:REAL;IN NumRuns:INTEGER;
          IN TotalBlocks:INTEGER);
 VAR
        5 LINES REMOVED
  MRRT,
  MART,
  TermCondition,
        9 LINES REMOVED
  MRRTMon,
  MARTMon,
  TermMon,
  GreenPercentMon,
  YellowPercentMon,
  RedPercentMon    : RStatObj;
  {statistical monitors which track the output variables from each run}
  TotalTime, TNodeCost, TBlockCost       : REAL;
  {the sum of green,yellow, and red time}
  outString       : LabelObj;
```

```
        32 LINES REMOVED
NEW(System);
NEW(SysDepGroup);
NEW(Strm1DepGroup);
NEW(Strm2DepGroup);
NEW(Strm3DepGroup);
NEW(SparesTrackingGroup);
NEW(SystemTrigger);
NEW(Stream1Trigger);
NEW(Stream2Trigger);
NEW(Stream3Trigger);
NEW(SysStreamA);
NEW(SysStreamB);
NEW(temp3);
NEW (FailStream,1..NumberOfBlocks);
NEW (RepairStream,1..NumberOfBlocks);
NEW (ARepairStream,1..NumberOfBlocks);
NEW(Block,1..NumberOfBlocks);
        15 LINES REMOVED
    NEW (ARepairStream[block.objectNum]);
    ASK ARepairStream[block.objectNum] TO SetSeed(FetchSeed(Block[block.objectNum].ARepairSeedNumber));
    {sets the seed numbers for components with independent random number streams}
  END IF;
END FOR;
FOR i:=1 TO totalNodes
        41 LINES REMOVED
CompRepairMon:=GETMONITOR(CompTimeToRepair,RStatObj);
CompRRepairMon:=GETMONITOR(CompTimeToRRepair,RStatObj);
CompARepairMon:=GETMONITOR(CompTimeToARepair,RStatObj);
GreenMon:=GETMONITOR(GreenTime,RStatObj);
YellowMon:=GETMONITOR(YellowTime,RStatObj);
RedMon:=GETMONITOR(RedTime,RStatObj);
AvailMon:=GETMONITOR(Availability,RStatObj);
TNodeMon:=GETMONITOR(TotalNodeCost,RStatObj);
TBlockMon:= GETMONITOR(TotalBlockCost,RStatObj);
{Cost monitors to calculate node and block cost for each simulation run.}
MTBDEMon:=GETMONITOR(MTBDE,RStatObj);
MDTMon:=GETMONITOR(MDT,RStatObj);
MTBMCMon:=GETMONITOR(MTBMC,RStatObj);
MRTMon:=GETMONITOR(MRT,RStatObj);
MRRTMon:=GETMONITOR(MRRT,RStatObj);
MARTMon:=GETMONITOR(MART,RStatObj);
TermMon:=GETMONITOR(TermCondition,RStatObj);
GreenPercentMon:=GETMONITOR(GreenPercent,RStatObj);
        191 LINES REMOVED
      Availability:=1.0;
    END IF;
    {IF (TNodeMon.Count>0)
      TNodeCost := TNodeMon.Sum/FLOAT(TNodeMon.Count);
    ELSE
     TNodeCost := 0.0;
    END IF;
    IF (TBlockMon.Count>0)
      TotBlockCost := TBlockCost;
    ELSE
     TBlockCost := 0.0;
    END IF;      {calculates operational availability}}
    IF (RedMon.Count>0)
        14 LINES REMOVED
      MRT:=CompRepairMon.Mean;
      MRRT:=CompRRepairMon.Mean;
      MART:=CompARepairMon.Mean;
      {calculates the average of all the component repair times}
    ELSE
```

```
        17 LINES REMOVED
    {for failure terminated sims, the ending sim time is reported}
    END IF;
    DoEndOfRunUpdate(Availability, MTBDE, MDT, MTBMC, MRT,MRRT,MART,
                GreenPercent, YellowPercent, RedPercent,TNodeCost,TBlockCost);
    {calls a procedure that does an "end of run" update}
  ELSE
    IF EventsFile
        10 LINES REMOVED
  ASK CompRepairMon TO Reset;
  ASK CompRRepairMon TO Reset;
  ASK CompARepairMon TO Reset;
  ASK GreenMon TO Reset;
        24 LINES REMOVED
  IF KeepSimStats
  DoEndOfSimUpdate(NumRuns, AvailMon, MTBDEMon, MDTMon, MTBMCMon, MRTMon,MRRTMon,MARTMon,
          GreenPercentMon, YellowPercentMon, RedPercentMon, TermMon,TNodeMon, TBlockMon);
  {this call a procudure to accomplish end of simulation steps}
  simulated := TRUE;
        72 LINES REMOVED
FOREACH Group1Member IN Strm1DepGroup
  ASK Strm1DepGroup TO RemoveThis(Group1Member);
END FOREACH;
FOREACH Group2Member IN Strm2DepGroup
  ASK Strm2DepGroup TO RemoveThis(Group2Member);
END FOREACH;
FOREACH Group3Member IN Strm3DepGroup
  ASK Strm3DepGroup TO RemoveThis(Group3Member);
END FOREACH;
{removes blocks for SysDepGroup}
FOREACH GroupMember IN SparesTrackingGroup
        6 LINES REMOVED
    DISPOSE(RepairStream[i]);
    DISPOSE(ARepairStream[i]);
  END IF;
        8 LINES REMOVED
ASK SysDepGroup TO ObjTerminate;
ASK Strm1DepGroup TO ObjTerminate;
ASK Strm2DepGroup TO ObjTerminate;
ASK Strm3DepGroup TO ObjTerminate;
ASK SparesTrackingGroup TO ObjTerminate;
ASK SystemTrigger TO ObjTerminate;
ASK Stream1Trigger TO ObjTerminate;
ASK Stream2Trigger TO ObjTerminate;
ASK Stream3Trigger TO ObjTerminate;
        6 LINES REMOVED
DISPOSE(ARepairStream);
DISPOSE(SysStreamA);
        8 LINES REMOVED
ASK MRTMon TO Reset;
ASK MRRTMon TO Reset;
ASK MARTMon TO Reset;
        10 LINES REMOVED
        OPEN OUT FILES INFORMATION
        25 LINES REMOVED
NEW(LStream);
ASK LStream TO Open("C:\Raptor3c\Level.out", Output);
counta := 0;
countb := 0;
countc := 0;
countd := 0;
counte := 0;
countg := 0;
countf := 0;
NEW(NCStream);
ASK NCStream TO Open("C:\Raptor3c\NCost.out", Output);
```

```
 NEW(BCStream);
 ASK BCStream TO Open("C:\Raptor3c\BCost.out", Output);
 {Both node and cost output files are opened, they do not print out to the screen
 but are automatically put into these file names.}
          43 LINES REMOVED
  IF MRRTFile
  NEW(MRRTStream);
  SaveAsFile(nameOfFile, pathName, filter, "MRRT Data File Name");
  IF nameOfFile = "NoFile"
   MRRTFile := FALSE;
   DISPOSE(MRRTStream);
  ELSE
   ASK MRRTStream TO Open(pathName + nameOfFile, Output);
  END IF;
 END IF;
 IF MARTFile
  NEW(MARTStream);
  SaveAsFile(nameOfFile, pathName, filter, "MART Data File Name");
  IF nameOfFile = "NoFile"
   MARTFile := FALSE;
   DISPOSE(MARTStream);
  ELSE
   ASK MARTStream TO Open(pathName + nameOfFile, Output);
  END IF;
 END IF;
          76 LINES REMOVED
          CLOSE OUT FILES INFORMATION
PROCEDURE CloseOutFiles;
BEGIN
 ASK LStream TO Close;
 DISPOSE(LStream);
 ASK NCStream TO Close;
 DISPOSE(NCStream);
 ASK BCStream TO Close;
 DISPOSE(BCStream);
 {Close out both the node and the block cost files.}
          18 LINES REMOVED
IF MRTFile
  ASK MRTStream TO Close;
  DISPOSE(MRTStream);
 END IF;
 IF MRRTFile
  ASK MRRTStream TO Close;
  DISPOSE(MRRTStream);
 END IF;
 IF MARTFile
  ASK MARTStream TO Close;
  DISPOSE(MARTStream);
 END IF;
          26 LINES REMOVED
          DO END OF RUN UPDATE INFORMATION
PROCEDURE DoEndOfRunUpdate(IN Availability, MTBDE, MDT, MTBMC, MRT,MRRT,MART,
              GreenPercent, YellowPercent, RedPercent, TNodeCost,TBlockCost : REAL);
VAR
          9 LINES REMOVED
  numAllocated,numlevone, numlevtwo,numlevthree, numlevfour, numlevfive      : INTEGER;
  tempnodecost, temptime,tempdonecost : REAL;
  tempblockcost, temptime2, tempdonecost2 : REAL;
BEGIN
  tempnodecost := 0.0;
  tempdonecost := 0.0;
  tempblockcost := 0.0;
  tempdonecost2 := 0.0;
  numlevone := 0;
  numlevtwo := 0;
  numlevthree := 0;
```

```
numlevfour := 0;
numlevfive := 0;
FOR i:=1 TO NumberOfBlocks
 IF (Block[i].Type=Component)AND(NOT(Block[i].EventBlock))
  IF (Block[i].Status=Running)
    temptime2 := SimTime - Block[i].LastBlockColorChange;
    tempdonecost2 := (temptime2*Block[i].UCost);
  ELSIF (Block[i].Status=Repairing)
    temptime2 := SimTime - Block[i].LastBlockColorChange;
    tempdonecost2 := (temptime2*Block[i].DCost);
  ELSIF (Block[i].Status=Idle)
    temptime2 := SimTime - Block[i].LastBlockColorChange;
    tempdonecost2 := (temptime2*Block[i].SCost);
  END IF;
  tempblockcost := tempdonecost2 + tempblockcost + Block[i].BlockCost + Block[i].ICost;
 END IF;
 {Calculates the cost of all the operating blocks by determining their final condition
 and adding this value into their running cost total.}
 IF (Block[i].Type=Component)AND(Block[i].EventBlock)
   tempblockcost := tempblockcost + Block[i].BlockCost + Block[i].ICost;
 END IF;
 IF (Block[i].EventBlock)
  IF ((TRUNC(Block[i].LFactor)=1) AND (Block[i].Status=Failure))
    numlevone := numlevone + 1;
  ELSIF ((TRUNC(Block[i].LFactor)=2) AND (Block[i].Status=Failure))
    numlevtwo := numlevtwo + 1;
  ELSIF ((TRUNC(Block[i].LFactor)=3) AND (Block[i].Status=Failure))
    numlevthree := numlevthree + 1;
  ELSIF ((TRUNC(Block[i].LFactor)=4) AND (Block[i].Status=Failure))
    numlevfour := numlevfour + 1;
  ELSIF ((TRUNC(Block[i].LFactor)=5) AND (Block[i].Status=Failure))
    numlevfive := numlevfive + 1;
  END IF;
 END IF;
 {Calculates the cost for the event blocks.}
 IF (Block[i].Type=Node)
  IF (Block[i].Status=Done)
    temptime := SimTime - Block[i].LastNodeColorChange;
    tempdonecost := (temptime*Block[i].Rcost);
  END IF;
  tempnodecost := tempdonecost + tempnodecost + Block[i].NodeCost;
 END IF;
 {Calculates the node cost for all nodes except the stop node.}
 IF (Block[i].Type=Stop)
  IF (Block[i].Status=Done)
    temptime := SimTime - Block[i].LastNodeColorChange;
    tempdonecost := (temptime*Block[i].Rcost);
  END IF;
  tempnodecost := tempnodecost + tempdonecost + Block[i].NodeCost;
 END IF;
 {Calculates the stop node cost.}
END FOR;
TNodeCost := tempnodecost;
TBlockCost := tempblockcost;
TotalNodeCost := TNodeCost;
TotalBlockCost := TBlockCost;
IF (tempstat)
 {ASK LStream TO WriteString("Run number " + INTTOSTR(runNumber));
 ASK LStream TO WriteLn;}
 IF (numlevone>0)
  {ASK LStream TO WriteString("Number of level one failures " + INTTOSTR(numlevone));
  ASK LStream TO WriteLn;                                    }
  counta := counta + 1;
 END IF;
 IF (numlevtwo>1)
  {ASK LStream TO WriteString("Number of level two failures " + INTTOSTR(numlevtwo));
```

```
    ASK LStream TO WriteLn;}
    countb := countb + 1;
   END IF;
   IF (numlevthree>8);
    {ASK LStream TO WriteString("Number of level three failure " + INTTOSTR(numlevthree));
    ASK LStream TO WriteLn;}
    countc := countc + 1;
   END IF;
   IF (numlevfour>1)
    {ASK LStream TO WriteString("Number of level four failures " + INTTOSTR(numlevfour));
    ASK LStream TO WriteLn;}
    countd := countd + 1;
   END IF;
   IF (numlevfive>4);
    {ASK LStream TO WriteString("Number of level five failure " + INTTOSTR(numlevfive));
    ASK LStream TO WriteLn;}
    counte := counte + 1;
   END IF;
   IF (numlevtwo=1) AND (numlevthree>2)
    {ASK LStream TO WriteString("Combination 1 failure");
    ASK LStream TO WriteLn;}
    countf := countf + 1;
   END IF;
   IF (numlevfour=1) AND (numlevfive>0)
    {ASK LStream TO WriteString("Combination 2 failure");
    ASK LStream TO WriteLn;}
    countg := countg + 1;
   END IF;
  END IF;
  {Update the statistics being collected on cost.}
IF ShowEORstats
          60 LINES REMOVED
       ASK outString TO SetLabel(SUBSTR(1, (p - 1), REALTOSTR(MRT)));
   END IF;
    outString := ASK statusBox Child("MRRTLabel", 750);
   labelString := SUBSTR(1, 6, REALTOSTR(MRRT));
   p := POSITION(labelString, ".");
   IF p <> 0
     ASK outString TO SetLabel(labelString);
   ELSE
     p := POSITION(REALTOSTR(MRRT), ".");
     ASK outString TO SetLabel(SUBSTR(1, (p - 1), REALTOSTR(MRRT)));
   END IF;
   outString := ASK statusBox Child("MARTLabel", 751);
   labelString := SUBSTR(1, 6, REALTOSTR(MART));
   p := POSITION(labelString, ".");
   IF p <> 0
     ASK outString TO SetLabel(labelString);
   ELSE
     p := POSITION(REALTOSTR(MART), ".");
     ASK outString TO SetLabel(SUBSTR(1, (p - 1), REALTOSTR(MART)));
   END IF;
  ELSE
   outString := ASK statusBox Child("MTBMLabel", 706);
          8 LINES REMOVED
   outString := ASK statusBox Child("MRTLabel", 707);
   ASK outString TO SetLabel("n/a");
   outString := ASK statusBox Child("MRRTLabel", 750);
   ASK outString TO SetLabel("n/a");
   outString := ASK statusBox Child("MARTLabel", 751);
   ASK outString TO SetLabel("n/a");
  END IF;
  ASK statusBox TO Draw;
END IF;
  ASK NCStream TO WriteReal(TNodeCost, 15,6);
  ASK NCStream TO WriteLn;
```

```
  ASK BCStream TO WriteReal(TBlockCost, 15,6);
  ASK BCStream TO WriteLn;
  {Writes out the individual run cost for both node total and block total.}
  IF AvailFile
          27 LINES REMOVED
  {writes the calculated value for MRT for this run to a file}
  END IF;
  IF MRRTFile AND (CompFailMon.Count>0)
  {IF the MRT box was checked in sim options dialog box}
    ASK MRRTStream TO WriteReal(MRRT,12,6);
    ASK MRRTStream TO WriteLn;
    {writes the calculated value for MRT for this run to a file}
  END IF;
  IF MARTFile AND (CompFailMon.Count>0)
  {IF the MRT box was checked in sim options dialog box}
    ASK MARTStream TO WriteReal(MART,12,6);
    ASK MARTStream TO WriteLn;
    {writes the calculated value for MRT for this run to a file}
  END IF;
  IF EndSimTimeFile
          313 LINES REMOVED, THE REMAINDER OF THE PROCEDURE
          DO END OF SIM UPDATE INFORMATION
PROCEDURE DoEndOfSimUpdate(IN NumRuns : INTEGER;
                IN AvailMon, MTBDEMon, MDTMon, MTBMCMon, MRTMon, MRRTMon, MARTMon,
                  GreenPercentMon,
                  YellowPercentMon, RedPercentMon, TermMon, TNodeMon, TBlockMon : RStatObj);
  VAR
          24 LINES REMOVED
    ASK LStream TO WriteString("Level one " + INTTOSTR(counta));
    ASK LStream TO WriteLn;
    ASK LStream TO WriteString("Level two " + INTTOSTR(countb));
    ASK LStream TO WriteLn;
    ASK LStream TO WriteString("Level three " + INTTOSTR(countc));
    ASK LStream TO WriteLn;
    ASK LStream TO WriteString("Level four " + INTTOSTR(countd));
    ASK LStream TO WriteLn;
    ASK LStream TO WriteString("Level five " + INTTOSTR(counte));
    ASK LStream TO WriteLn;
    ASK LStream TO WriteString("Combo one " + INTTOSTR(countf));
    ASK LStream TO WriteLn;
    ASK LStream TO WriteString("Combo two " + INTTOSTR(countg));
    ASK LStream TO WriteLn;
    ASK outString TO Draw;
          35 LINES REMOVED
    NEW(FinalArray, 1..6, 1..(14 + SparesTrackingGroup.numberIn + totalPools));
    NEW(FinCostArray, 1..2, 1..4);
    {creates a big array of string used in print and screen display routines}
    IF TimeTerminated
          54 LINES REMOVED
    FinCostArray[1,1] := REALTOSTR(TNodeMon.Mean);
    FinCostArray[2,1] := REALTOSTR(TBlockMon.Mean);
    FinCostArray[1,2] := REALTOSTR(TNodeMon.Minimum);
    FinCostArray[2,2] := REALTOSTR(TBlockMon.Minimum);
    FinCostArray[1,3] := REALTOSTR(TNodeMon.Maximum);
    FinCostArray[2,3] := REALTOSTR(TBlockMon.Maximum);
    {Defines the overall simulation cost calculations and puts them in a matrix to be
    output to the cost files, for nodes and blocks.}
    IF (TNodeMon.Count>1)
      FinCostArray[1,4] := REALTOSTR(TNodeMon.StdDev);
    ELSE
      FinCostArray[1,4] := "n/a";
    END IF;
    IF (TBlockMon.Count>1)
      FinCostArray[2,4] := REALTOSTR(TBlockMon.StdDev);
    ELSE
      FinCostArray[2,4] := "n/a";
```

```
  END IF;
  IF (AvailMon.Count>1)
        36 LINES REMOVED
    FinalArray[1,6] := "MRT";
    FinalArray[1,7] := "MRRT";
    FinalArray[1,8] := "MART";
  ELSE
    FinalArray[2,5]:=">"+REALTOSTR(MTBMCMon.Mean);
    FinalArray[4,5]:=">"+REALTOSTR(MTBMCMon.Maximum);
    FinalArray[5,5]:="n/a";
    FinalArray[1,6]:="MRT ("+INTTOSTR(MRTMon.Count)+" runs)";
    FinalArray[1,7]:="MRRT ("+INTTOSTR(MRRTMon.Count)+" runs)";
    FinalArray[1,8]:="MART ("+INTTOSTR(MARTMon.Count)+" runs)";
  END IF;
  FinalArray[3,5]:=REALTOSTR(MTBMCMon.Minimum);
  FinalArray[2,6]:=REALTOSTR(MRTMon.Mean);
  FinalArray[3,6]:=REALTOSTR(MRTMon.Minimum);
  FinalArray[4,6]:=REALTOSTR(MRTMon.Maximum);
  FinalArray[2,7]:=REALTOSTR(MRRTMon.Mean);
  FinalArray[3,7]:=REALTOSTR(MRRTMon.Minimum);
  FinalArray[4,7]:=REALTOSTR(MRRTMon.Maximum);
  FinalArray[2,8]:=REALTOSTR(MARTMon.Mean);
  FinalArray[3,8]:=REALTOSTR(MARTMon.Minimum);
  FinalArray[4,8]:=REALTOSTR(MARTMon.Maximum);
  IF (MRTMon.Count>1)
    FinalArray[5,6]:=REALTOSTR(MRTMon.StdDev);
    FinalArray[5,7]:=REALTOSTR(MRRTMon.StdDev);
    FinalArray[5,8]:=REALTOSTR(MARTMon.StdDev);
  ELSE
    FinalArray[5,6]:="n/a";
    FinalArray[5,7]:="n/a";
    FinalArray[5,8]:="n/a";
  END IF;
  FinalArray[2,9]:=REALTOSTR(GreenPercentMon.Mean);
        103 LINES REMOVED
    DisplayResults;
  ASK NCStream TO WriteString("mean " + FinCostArray[1, 1] + " min " + FinCostArray[1, 2]
    + " max " + FinCostArray[1, 3] + " sd " + FinCostArray[1, 4]);
  ASK NCStream TO WriteLn;
  ASK BCStream TO WriteString("mean " + FinCostArray[2, 1] + " min " + FinCostArray[2, 2]
    + " max " + FinCostArray[2, 3] + " sd " + FinCostArray[2, 4]);
  ASK BCStream TO WriteLn;
  {Outputs the node and block costs to the predefined files.}
  IF FinalFile
    PrintResults(finalReportFile);
  END IF;
END PROCEDURE; {DoEndOfSimUpdate}
        GET SIM OPTIONS INFORMATION
PROCEDURE GetSimOptions(IN c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c22,
            c23,c24: BOOLEAN);
        10 LINES REMOVED
 MRRTFile:=c23;
 MARTFile:=c24;
 SysFailTimeFile:=c7;
        11 LINES REMOVED
        POOL OBJECT INFORMATION
OBJECT PoolObj;
        METHOD GENERATE SPARE REMOVED
        ORDER A SPARE INFORMATION
 TELL METHOD OrderASpare(IN i:INTEGER);
        32 LINES REMOVED
    END IF;
    IncrementResourcesBy(TRUNC(PoolArray[i].emergencyNumber));
    PoolResAvailable[i]:=Resources;
 END METHOD;  {OrderASpare}
 END OBJECT;
```

```
          BLOCK OBJ INFORMATION
OBJECT BlockObj;
          RECONFIGURE FAIL INFORMATION
 ASK METHOD ReconfigureFail();
   VAR
    NextBlock   : INTEGER;
    {a block immediately downstream of the block which calls this routine}
    NextBlockType: BlockType;
    {either component, start, stop, or node}
    j,k        : INTEGER;
    dBoxLabel   :LabelObj;
   BEGIN
    IF (StreamDependence=1)
      IF SystemDependence
        SysStrm := 3;
      ELSE
        SysStrm := 1;
      END IF;
      FOREACH Group1Member IN Strm1DepGroup
           {runs this loop for each system dependent block}
          IF ((Group1Member.Status=Running) OR (Group1Member.Status=UsingCold))
            {if currently operating}
            IF Group1Member.SparingMethod=ColdPool
              Interrupt(Group1Member,"OperateWithColdStandbys");
            ELSE
              Interrupt(Group1Member,"Operate");
            END IF;
            {halts operation, starts that blocks ON INTERRUPT clause}
          END IF;
       END FOREACH;
    END IF;
    {Handles manipulation of the stream dependent group, in this case, group 1.}
    IF (StreamDependence=2)
      IF SystemDependence
        SysStrm := 3;
      ELSE
        SysStrm := 1;
      END IF;
      FOREACH Group2Member IN Strm2DepGroup
           {runs this loop for each system dependent block}
          IF ((Group2Member.Status=Running) OR (Group2Member.Status=UsingCold))
            {if currently operating}
            IF Group2Member.SparingMethod=ColdPool
              Interrupt(Group2Member,"OperateWithColdStandbys");
            ELSE
              Interrupt(Group2Member,"Operate");
            END IF;
            {halts operation, starts that blocks ON INTERRUPT clause}
          END IF;
       END FOREACH;
    END IF;
     {Handles manipulation of the stream dependent group, in this case, group 2.}
    IF (StreamDependence=3)
      IF SystemDependence
        SysStrm := 3;
      ELSE
        SysStrm := 1;
      END IF;
      FOREACH Group3Member IN Strm3DepGroup
           {runs this loop for each system dependent block}
          IF ((Group3Member.Status=Running) OR (Group3Member.Status=UsingCold))
            {if currently operating}
            IF Group3Member.SparingMethod=ColdPool
              Interrupt(Group3Member,"OperateWithColdStandbys");
            ELSE
              Interrupt(Group3Member,"Operate");
```

```
          END IF;
          {halts operation, starts that blocks ON INTERRUPT clause}
        END IF;
      END FOREACH;
    END IF;
    {Handles manipulation of the stream dependent group, in this case, group 3.}
    NumGoodPaths:=NumGoodPaths-1;
    {decreases the number of good paths entering the block}
    IF (NumGoodPaths=GoodPathsRequired-1)
    {if this block now has one less than needed to perform its function}
      IF (Type=Node)
        NodeTimeToFail := SimTime - LastNodeColorChange;
        IF (Status=Running)
          NodeCost := NodeCost + (NodeTimeToFail*Gcost);
        END IF;
        IF (Status=Degraded)
          NodeCost := NodeCost + (NodeTimeToFail*Ycost);
        END IF;
        {Calculates the node cost by analyzing the last condition of each node and
        multiplying by the appropriate cost factor.}
        ChangeStatus(Repairing);
        LastNodeColorChange := SimTime;
        {this will cause nodes to turn red, but allows components to remain green
         while allowing the propagation to continue}
      END IF;
      IF (Type=Stop)
        BlockTimeToFail := SimTime - LastBlockColorChange;
        BlockCost := BlockCost + (BlockTimeToFail*Gcost);
        {Calculates the stop node cost for the previous green time.}
        LastBlockColorChange := SimTime;
        ASK System TO ChangeStatus(RRed);
        IF (SystemDependence) AND (StreamDependence=0)
          SysStrm := 2;
        END IF;
        IF (SystemDependence) AND (StreamDependence>0)
          SysStrm := 3;
        END IF;
        {calls a routine which lets the system know it is down}
        SysTimeToFail:=SimTime-LastUpDownChange;
          48 LINES REMOVED
      IF (Type=Node) AND (Status=Running)
        NodeTimeToFail := SimTime - LastNodeColorChange;
        NodeCost := NodeCost + (NodeTimeToFail*Gcost);
        LastNodeColorChange := SimTime;
        {Calculates the last portion of green node cost using the appropriate cost factor.}
        ChangeStatus(Degraded);
        {if the block is a node and the failure did not cause the node to lose
         functionality, change the node to yellow indicating at least one input
         path is down}
      END IF;
    END IF;
    IF (System.Status=RRed) {if failure takes system down, set priority higher}
      spcResPriority:=2.0;
    ELSE
      spcResPriority:=1.0;
    END IF;
    {This defines the priority given to each block to determine the order of repair
    when limited resources are used, if the failure of the given block takes down the system,
    then the block is given a higher priority.}
END METHOD;  {ReconfigureFail}
          RECONFIGURE REPAIR INFORMATION
ASK METHOD ReconfigureRepair();
          14 LINES REMOVED
          NodeTimeToFail := SimTime - LastNodeColorChange;
          IF (Status=Repairing)
          NodeCost := NodeCost + (NodeTimeToFail*Rcost);
```

```
        END IF;
        IF (Status=Degraded)
        NodeCost := NodeCost + (NodeTimeToFail*Ycost);
        END IF;
        {Handles the node cost calculations using the appropriate cost factors.}
        LastNodeColorChange := SimTime;
        ChangeStatus(Running);
        {the node is fully functional}
      ELSE
      {the node is repaired but still has some inbound links that are down}
        NodeTimeToFail := SimTime - LastNodeColorChange;
        IF (Status=Repairing)
        NodeCost := NodeCost + (NodeTimeToFail*Rcost);
        END IF;
        IF (Status=Degraded)
        NodeCost := NodeCost + (NodeTimeToFail*Ycost);
        END IF;
        {Handles the node cost calculations using the appropriate cost factors.}
        LastNodeColorChange := SimTime;
        ChangeStatus(Degraded);
        {changes the node's status to yellow}
      END IF;
    END IF;
    IF (Type=Stop)
      NumSysFailures:=NumSysFailures+1;
      SysTimeToRepair:=SimTime-LastUpDownChange;
      {determines the duration of the system down time}
      BlockTimeToFail := SimTime - LastBlockColorChange;
      BlockCost := BlockCost + (BlockTimeToFail*DCost);
      {Updating the node cost calculations.}
      LastBlockColorChange := SimTime;
        16 LINES REMOVED
        IF (NumSysFailures=TRUNC(StartStats))
          TELL System ResetAllStatistics;
          FOR i:=1 TO totalNodes
            node := ASK root Child("RBDNode", i);
            TELL Block[node.objectNum] TO ResetNodeStatistics;
          END FOR;
          FOR i := 1 TO NumberOfBlocks
        38 LINES REMOVED
    IF (Type=Node) AND (NumGoodPaths=NumPathsIn)
      NodeTimeToFail := SimTime - LastNodeColorChange;
      IF (Status=Repairing)
        NodeCost := NodeCost + (NodeTimeToFail*Rcost);
      END IF;
      IF (Status=Degraded)
        NodeCost := NodeCost + (NodeTimeToFail*Ycost);
      END IF;
      {More cost calculations to determine the overall node cost per simulation run.}
      LastNodeColorChange := SimTime;
      ChangeStatus(Running);
      {changes the node to green if all inbound paths are up}
    END IF;
  END IF;
  IF (StreamDependence=1)
  ASK Stream1Trigger TO Release;
  END IF;
  IF (StreamDependence=2)
    ASK Stream2Trigger TO Release;
  END IF;
  IF (StreamDependence=3)
    ASK Stream3Trigger TO Release;
  END IF;
END METHOD;   {ReconfigureRepair}
        RECONFIGURE SYSTEM INFORMATION
 ASK METHOD ReconfigureSystem();
```

```
          10 LINES REMOVED
      IF (Type=Node)
        NodeTimeToFail := SimTime - LastNodeColorChange;
        IF (Status=Repairing)
          NodeCost := NodeCost + (NodeTimeToFail * Rcost);
        END IF;
        IF (Status=Degraded)
          NodeCost := NodeCost + (NodeTimeToFail * Ycost);
        END IF;
        IF (Status=Running)
          NodeCost := NodeCost + (NodeTimeToFail * Gcost);
        END IF;
        LastNodeColorChange := SimTime;
        ChangeStatus(Done);
          46 LINES REMOVED
          OPERATE INFORMATION
TELL METHOD Operate(IN i:INTEGER);
  VAR
    TimeToFail,
    OperatingTime,
    TimeToRepair,
    TimeToARepair,
    ProjectedFailTime,
    TempFactor        : REAL;
    waitCompleted,
    TriggerFired,
    TriggerStrmFired,
    Trigger1Fired,
    Trigger2Fired,
    Trigger3Fired,
    SpareAvailable    : BOOLEAN;
    statusString,
    printString,
    outString         : STRING;
    j                 : INTEGER;
  BEGIN
    WHILE (Status=Running)
    {this loop prevents permanently failed blocks from reentering the loop}
  {***Draw a random number for time to failure***}
      IF (FailSeedNumber=9)
        ASK SysStreamA TO DrawNumber(FailDist,FailDistParam,TimeToFail);
        {this draws a random number from system stream A}
      ELSIF (FailSeedNumber=10)
        ASK SysStreamB TO DrawNumber(FailDist,FailDistParam,TimeToFail);
        {this draws a random number from system stream B}
      ELSE
        ASK FailStream[i] TO DrawNumber(FailDist,FailDistParam,TimeToFail);
        {this draws a random number from an independent stream}
      END IF;
      {the variable TimeToFail is returned from the DrawNumber routine}
   {*** wait until component fails***}
      IF (TimeToFail<0.0)
        GetErrorBox("56: Negative failure time rounded to zero!");
        TimeToFail:=0.0;
      END IF;
      ProjectedFailTime:=SimTime+TimeToFail;
      OperatingTime:=TimeToFail;
      waitCompleted:=FALSE;
      WHILE waitCompleted=FALSE
        IF (FailureCounter=0)
          TimeToFail := TimeToFail;
          {First failure, the failure time is not affected.}
        ELSIF (RRAdjfactor = 1.0) AND (RAdjfactor = 1.0)
          TimeToFail := TimeToFail;
          {Neither the remove and replace or repair have an growth/degradation factor.}
        ELSIF (RRAdjfactor>1.0) AND (RAdjfactor = 1.0)
```

```
      TempFactor :=  (((RRAdjfactor-1.0)*FLOAT(RRFailureCounter))+1.0);
      IF TempFactor > RRLimfactor
        TimeToFail := RRLimfactor*TimeToFail;
      ELSE
        TimeToFail := TempFactor*TimeToFail;
      END IF;
      {Handles the case of remove and replace reliability growth.}
      {This does not include repair growth/degradation factor.}
ELSIF (RRAdjfactor<1.0) AND (RAdjfactor = 1.0)
   TempFactor := 1.0;
   FOR j := 1 TO RRFailureCounter
     TempFactor := RRAdjfactor*TempFactor;
   END FOR;
   IF TempFactor < RRLimfactor
       TimeToFail := RRLimfactor*TimeToFail;
    ELSE
       TimeToFail := TempFactor*TimeToFail;
    END IF;
    {Handles the case of remove and replace reliabilty degradation.
    Does not include repair growth/degradation.}
ELSIF (RRAdjfactor=1.0) AND (RAdjfactor>1.0)
   IF (RFailureCounter=0)
     TimeToFail := TimeToFail;
   ELSE
      TempFactor :=  (((RAdjfactor-1.0)*FLOAT(RFailureCounter))+1.0);
      IF TempFactor > RLimfactor
        TimeToFail := RLimfactor*TimeToFail;
      ELSE
        TimeToFail := TempFactor*TimeToFail;
      END IF;
   END IF;
    {Handles the case of repair reliabilty growth.
    Does not include remove and replace growth/degradation.}
ELSIF (RRAdjfactor=1.0) AND (RAdjfactor<1.0)
   IF (RFailureCounter=0)
     TimeToFail  := TimeToFail;
   ELSE
     TempFactor := 1.0;
     FOR j := 1 TO RFailureCounter
     TempFactor := RAdjfactor*TempFactor;
     END FOR;
     IF TempFactor < RLimfactor
        TimeToFail := RLimfactor*TimeToFail;
     ELSE
       TimeToFail := TempFactor*TimeToFail;
     END IF;
   END IF;
   {Handles the case of repair degradation, does not include
   remove and replace growth/degradation.}
ELSIF (RRAdjfactor>1.0) AND (RAdjfactor>1.0)
   TempFactor := (((RRAdjfactor-1.0)*FLOAT(RRFailureCounter))+1.0);
   IF TempFactor > RRLimfactor
     TimeToFail := RRLimfactor*TimeToFail;
   ELSE
     TimeToFail := TempFactor*TimeToFail;
   END IF;
   IF (RFailureCounter=0)
     TimeToFail := TimeToFail;
   ELSE
     TempFactor :=(((RAdjfactor-1.0)*FLOAT(RFailureCounter))+1.0);
     IF TempFactor > RLimfactor
       TimeToFail := RLimfactor*TimeToFail;
     ELSE
       TimeToFail := TempFactor*TimeToFail;
     END IF;
   END IF;
```

```
    {Handles the case when remove and replace is improving the system,
     along with repair maintenance is improving the system.}
ELSIF (RRAdjfactor>1.0) AND (RAdjfactor<1.0)
   TempFactor := (((RRAdjfactor-1.0)*FLOAT(RRFailureCounter))+1.0);
   IF TempFactor > RRLimfactor
     TimeToFail := RRLimfactor*TimeToFail;
   ELSE
     TimeToFail := TempFactor*TimeToFail;
   END IF;
   IF (RFailureCounter=0)
     TimeToFail := TimeToFail;
   ELSE
     TempFactor := 1.0;
    FOR j := 1 TO RFailureCounter
     TempFactor := RAdjfactor*TempFactor;
    END FOR;
    IF TempFactor < RLimfactor
       TimeToFail := RLimfactor*TimeToFail;
    ELSE
       TimeToFail := TempFactor*TimeToFail;
    END IF;
   END IF;
   {Handles the case when remove and replace is improving the system,
    and repair is degrading the system.}
ELSIF (RRAdjfactor<1.0) AND (RAdjfactor>1.0)
   TempFactor := 1.0;
   FOR j := 1 TO RRFailureCounter
     TempFactor := RRAdjfactor*TempFactor;
   END FOR;
   IF TempFactor < RRLimfactor
       TimeToFail := RRLimfactor*TimeToFail;
    ELSE
       TimeToFail := TempFactor*TimeToFail;
    END IF;
   IF (RFailureCounter=0)
     TimeToFail := TimeToFail;
   ELSE
     TempFactor :=(((RAdjfactor-1.0)*FLOAT(RFailureCounter))+1.0);
     IF TempFactor > RLimfactor
       TimeToFail := RLimfactor*TimeToFail;
     ELSE
       TimeToFail := TempFactor*TimeToFail;
     END IF;
   END IF;
   {Handles the case when remove and replace is degrading the system,
    and repair is improving the system.}
ELSE
   TempFactor := 1.0;
   FOR j := 1 TO RRFailureCounter
     TempFactor := RRAdjfactor*TempFactor;
   END FOR;
   IF TempFactor < RRLimfactor
       TimeToFail := RRLimfactor*TimeToFail;
    ELSE
       TimeToFail := TempFactor*TimeToFail;
    END IF;
   IF (RFailureCounter=0)
     TimeToFail := TimeToFail;
   ELSE
     TempFactor := 1.0;
    FOR j := 1 TO RFailureCounter
     TempFactor := RAdjfactor*TempFactor;
    END FOR;
    IF TempFactor < RLimfactor
       TimeToFail := RLimfactor*TimeToFail;
    ELSE
```

```
      TimeToFail := TempFactor*TimeToFail;
    END IF;
  END IF;
END IF;
{Handles the final case when both remove and replace and repair are
 degrading the system.}
WAIT DURATION TimeToFail;
  waitCompleted:=TRUE;
ON INTERRUPT
  IF (Status=Running)
    ChangeStatus(Idle);
    BlockCost := BlockCost + (SimTime-LastBlockColorChange)*UCost;
    LastBlockColorChange := SimTime;
  END IF;
  IF (SysStrm=2)
   TriggerFired:=FALSE;
   TriggerStrmFired := TRUE;
  ELSE
   TriggerStrmFired := FALSE;
   IF (SysStrm=3)
     TriggerFired := FALSE;
   ELSE
     TriggerFired := TRUE;
   END IF;
   IF (StreamDependence=1)
     Trigger1Fired := FALSE;
   ELSIF (StreamDependence=2)
     Trigger2Fired := FALSE;
   ELSE
     Trigger3Fired := FALSE;
   END IF;
  END IF;
  {Handles the stream dependent triggers to release the stream dependent components.}
  TimeToFail:=ProjectedFailTime-SimTime;
  WHILE ((TriggerFired=FALSE) OR (TriggerStrmFired=FALSE))
    WHILE TriggerFired=FALSE
     WAIT FOR SystemTrigger TO Fire;
       TriggerFired:=TRUE;
       IF (Status=Idle)
         ChangeStatus(Running);
         BlockCost := BlockCost + (SimTime-LastBlockColorChange)*SCost;
         LastBlockColorChange := SimTime;
       END IF;
     ON INTERRUPT
    {this is necessary to prevent a crash in the rare event that the system
    is repaired and then fails at the same time, before the end wait for this
    component is executed. (it usually happens when the operator makes a
    mistake and has identical components on identical independent streams)}
     END WAIT;
     TriggerStrmFired := TRUE;
    END WHILE;
    WHILE TriggerStrmFired=FALSE
     IF (StreamDependence=1)
       WAIT FOR Stream1Trigger TO Fire;
       TriggerStrmFired := TRUE;
       Trigger1Fired := TRUE;
       ON INTERRUPT
       END WAIT;
       IF ((SystemDependence=TRUE) AND (System.Status=RRed))
         ChangeStatus(Idle);
         BlockCost := BlockCost + (SimTime-LastBlockColorChange)*DCost;
         LastBlockColorChange := SimTime;
         WAIT FOR SystemTrigger TO Fire;
         TriggerFired := TRUE;
         IF (Status=Idle)
           ChangeStatus(Running);
```

```
            BlockCost := BlockCost + (SimTime-LastBlockColorChange)*SCost;
            LastBlockColorChange := SimTime;
          END IF;
          END WAIT;
        ELSE
          TriggerFired := TRUE;
        END IF;
{Handles manipulation of the stream 1 and system dependent components simultaneously.}
      ELSIF (StreamDependence=2)
        WAIT FOR Stream2Trigger TO Fire;
        TriggerStrmFired := TRUE;
        Trigger2Fired := TRUE;
        ON INTERRUPT
        END WAIT;
        IF ((SystemDependence=TRUE) AND (System.Status=RRed))
          ChangeStatus(Idle);
          BlockCost := BlockCost + (SimTime-LastBlockColorChange)*DCost;
          LastBlockColorChange := SimTime;
          WAIT FOR SystemTrigger TO Fire;
          TriggerFired := TRUE;
          IF (Status=Idle)
            ChangeStatus(Running);
            BlockCost := BlockCost + (SimTime-LastBlockColorChange)*SCost;
            LastBlockColorChange := SimTime;
          END IF;
          END WAIT;
        ELSE
          TriggerFired := TRUE;
        END IF;
{Handles manipulation of the stream 2 and system dependent components simultaneously.}
      ELSE
        WAIT FOR Stream3Trigger TO Fire;
        TriggerStrmFired := TRUE;
        Trigger3Fired := TRUE;
        ON INTERRUPT
        END WAIT;
        IF ((SystemDependence=TRUE) AND (System.Status=RRed))
          ChangeStatus(Idle);
          BlockCost := BlockCost + (SimTime-LastBlockColorChange)*DCost;
          LastBlockColorChange := SimTime;
          WAIT FOR SystemTrigger TO Fire;
          TriggerFired := TRUE;
          IF (Status=Idle)
            ChangeStatus(Running);
            BlockCost := BlockCost + (SimTime-LastBlockColorChange)*SCost;
            LastBlockColorChange := SimTime;
          END IF;
          END WAIT;
        ELSE
          TriggerFired := TRUE;
        END IF;
      END IF;
{Handles manipulation of the stream 3 and system dependent components simultaneously.}
      IF (Status=Idle)
        ChangeStatus(Running);
        BlockCost := BlockCost + (SimTime-LastBlockColorChange)*SCost;
        LastBlockColorChange := SimTime;
      END IF;
    END WHILE;
  END WHILE;
  ProjectedFailTime:=SimTime+TimeToFail;
 END WAIT;
END WHILE;
CompTimeToFail:=SimTime-TimeOfCompFailure;
{Left Monitored Stat variable keeping track of time between any failures}
BlockTimeToFail := SimTime-LastBlockColorChange;
```

```
BlockCost := BlockCost + BlockTimeToFail*UCost;
{Calculates the block cost for the overall simulation.}
LastBlockColorChange := SimTime;
TimeOfCompFailure:=SimTime;
{stores current sim time for use by CompTimeToFail above}
INC(NumCurrentlyFailed);
{increases by one a counter tracking the number of system components currently
failed}
temp4 := ASK temp3 TO UniformReal(0.0, 100.0);
 {Used to determine if a remove and replace or repair maintenance will occur.}
INC(FailureCounter);
IF (NOT(UseAdjust) OR (UseAdjust AND (temp4 > ProbAdjust)))
  INC(RRFailureCounter);
  RFailureCounter := 0;
ELSE
  INC(RFailureCounter);
END IF;
{Increments the appropriate failure counters.}
{increases a counter tracking the number of times this component has failed}
SpareAvailable:=TRUE;
BlockCost := BlockCost + SpareCost;
{Updates the block cost according to the spares required.}
ChangeStatus(Repairing);
  {changes component status to Red}
ReconfigureFail;
  {calls a routine to figure out the affect this failure has on the system}
IF (System.Status = GGreen)
{if all system components were green prior to this failure}
  ASK System TO ChangeStatus(YYellow);
  {changes system to yellow if this failure did not take down the system}
END IF;
IF EventsFile
{user selected event log from sim options dialog box}
  CASE System.Status
    WHEN GGreen:
      statusString:="Green";
    WHEN YYellow:
      statusString:="Yellow";
    WHEN RRed:
      statusString:="Red";
      {converts the system status to a string}
  END CASE;
  printString := REALTOSTR(SimTime) + "                ";
  printString := SUBSTR(1, 12, printString);
  outString:=printString;
  printString:= SUBSTR(1,20,BlockName + "                ");
  outString:=outString+printString;
  printString:= "  Failed ,  TimeOperated="+REALTOSTR(OperatingTime);
  outString:=outString+printString;
  printString:= "  System="+statusString;
  outString:=outString+printString;
  ASK EventStream TO WriteString(outString);
  {writes a line in the event log indicating the failure event}
  ASK EventStream TO WriteLn;
 END IF;   {EventsFile}
{***The next sections check to see if a spare is available for custom and pooled sparing
 blocks, if not then it checks to see if you can order one, if not it checks to see if
 any are ever going to come, if not the status is changed to "Done" and the block
 will no longer operate***}
 IF (NOT(UseAdjust) OR (UseAdjust AND (temp4 > ProbAdjust)))
 {The repair is just a typical remove and replace procedure.}
 IF (SparingMethod=Custom)
 {if the component does not have infinite spares}
   IF (CustomSpares[i].Resources=0)
   {if no spares are available}
     IF EventsFile
```

```
      {if user selected event log from sim options dialog box}
        printString := REALTOSTR(SimTime) + "                ";
        printString := SUBSTR(1, 12, printString);
        outString:=printString;
        printString:= SUBSTR(1,20,BlockName + "                ");
        outString:=outString+printString;
        printString:= "  ***No spare in stock*** ";
        outString:=outString+printString;
        ASK EventStream TO WriteString(outString);
        {write a line indicating component is out of spares}
        ASK EventStream TO WriteLn;
      END IF;
      IF (DelayForSpares>0.00001)
      {if spare ordering is allowed}
        OrderASpare(i);
        {call a routine to order the spare}
      ELSE
      {if you cannot order a spare}
        IF (ReplenishQuantity=0) OR (SpareReplenishTime<0.00001)
        {if you no spares are due to arrive later}
          IF EventsFile
          {user selected event log}
            printString := REALTOSTR(SimTime) + "                ";
            printString := SUBSTR(1, 12, printString);
            outString:=printString;
            printString:= SUBSTR(1,20,BlockName + "                ");
            outString:=outString+printString;
            printString:= "  ***permanently down, out of spares*** ";
            outString:=outString+printString;
            ASK EventStream TO WriteString(outString);
            {write a line in event log indicating component is out of spares}
            ASK EventStream TO WriteLn;
          END IF;
          ChangeStatus(Done);
          {call a routine which changes the color of the block to dark red}
          ReconfigureSystem;
          {propagates the permanent failure through the system}
        END IF;  {Any due to come in}
      END IF;  {Allowed to Order}
    END IF;  {Spare not available}
  END IF;  {Custom Spares}
END IF;
IF (NOT(UseAdjust) OR (UseAdjust AND (temp4 > ProbAdjust)))
 {The repair is just a typical remove and replace procedure.}
 IF (SparingMethod=SparePool)
    IF (PooledSpares[PoolNumber].Resources>0)
      intArray[PoolNumber] := 0;
    END IF;
    IF (PooledSpares[PoolNumber].Resources=0)
      IF EventsFile
      {if user selected event log from sim options dialog box}
        printString := REALTOSTR(SimTime) + "                ";
        printString := SUBSTR(1, 12, printString);
        outString:=printString;
        printString:= SUBSTR(1,20,BlockName + "                ");
        outString:=outString+printString;
        printString:= "  ***No spare in stock*** ";
        outString:=outString+printString;
        ASK EventStream TO WriteString(outString);
        {write a line indicating component is out of spares}
        ASK EventStream TO WriteLn;
      END IF;
       IF (PoolArray[PoolNumber].emergencyTime>0.00001)
         IF NOT(PoolArray[PoolNumber].emergencyFlag)
       {if spare ordering is allowed}
          IF intArray[PoolNumber] = 0
```

```
            TELL PooledSpares[PoolNumber] TO OrderASpare(PoolNumber);
            intArray[PoolNumber] := intArray[PoolNumber] + TRUNC(PoolArray[PoolNumber].emergencyNumber);
            intArray[PoolNumber] := intArray[PoolNumber] - 1;
           ELSE
            intArray[PoolNumber] := intArray[PoolNumber] - 1;
           END IF;
          END IF;
          {call a routine to order the spare}
         ELSE
         {if you cannot order a spare}
          IF (PoolArray[PoolNumber].newSpares=0) OR (PoolArray[PoolNumber].newSparesArrival<0.00001)
          {ifno spares are due to arrive later}
           ChangeStatus(Done);
           printString := REALTOSTR(SimTime) + "                ";
           printString := SUBSTR(1, 12, printString);
           outString:=printString;
           printString:= SUBSTR(1,20,BlockName + "                ");
           outString:=outString+printString;
           printString:= "  ***permanently down, out of spares*** ";
           outString:=outString+printString;
           IF EventsFile
           {user selected event log}
            ASK EventStream TO WriteString(outString);
            {write a line in event log indicating component is out of spares}
            ASK EventStream TO WriteLn;
           END IF;
           {call a routine which changes the color of the block to dark red}
           ReconfigureSystem;
           {propagates the permanent failure through the system}
          END IF;  {Any due to come in}
        END IF;  {Allowed to Order}
       END IF;  {Spare not available}
      END IF;  {SparePool}
     END IF;
     IF (NOT(UseAdjust) OR (UseAdjust AND (temp4 > ProbAdjust)))
     {The repair is just a remove and replace procedure.}
      IF (SparingMethod=None)
       ChangeStatus(Done);
       printString := REALTOSTR(SimTime) + "                ";
       printString := SUBSTR(1, 12, printString);
       outString:=printString;
       printString:= SUBSTR(1,20,BlockName + "                ");
       outString:=outString+printString;
       printString:= "  ***permanently down, out of spares*** ";
       outString:=outString+printString;
       IF EventsFile
       {user selected event log}
        ASK EventStream TO WriteString(outString);
        {write a line in event log indicating component is out of spares}
        ASK EventStream TO WriteLn;
       END IF;
       ReconfigureSystem;
       {propagates the permanent failure through the system}
      END IF;  {Sparing=None}
     END IF;
{***If is "Done" it will skip to the end of the procedure.***  }
      IF (Status=Repairing)
       WAIT DURATION LogisticsDelay;
       {waits the ALDT}
       END WAIT;
       BlockCost := BlockCost + LogisticsDelay*ALDTCost;
       {Update the block cost to reflect the logistic delay.}
       IF (NOT(UseAdjust)) OR (UseAdjust AND (temp4 > ProbAdjust))
       {The repair is just a normal remove and replace.}
       CASE SparingMethod
        WHEN Infinite:
```

```
    {};
  WHEN Custom:
    IF (CustomSpares[i].Resources=0)
      ChangeStatus(Hold);
    END IF;
    WAIT FOR CustomSpares[i] TO Give(SELF,1);
    {if a spare is available it gets it immediately, otherwise the componnent
    is stuck here until the spare arrives. The number of spares on hand
    is automatically reduced by one when the spare is used}
    END WAIT;
    IF Status=Hold
      ChangeStatus(Repairing);
    END IF;
    SparesAvailable[i]:=CustomSpares[i].Resources;
    {updates the stat variable keeping track of the number on hand}
  WHEN SparePool:
    IF  (PooledSpares[PoolNumber].Resources=0)
      ChangeStatus(Hold);
    END IF;
    WAIT FOR PooledSpares[PoolNumber] TO Give(SELF,1);
      PoolResAvailable[PoolNumber]:=PooledSpares[PoolNumber].Resources;
    END WAIT;
    IF (PooledSpares[PoolNumber].Resources = 0) AND (PoolArray[PoolNumber].emergencyFlag)
     TELL PooledSpares[PoolNumber] TO OrderASpare(PoolNumber);
    END IF;
    IF Status=Hold
      ChangeStatus(Repairing);
    END IF;
END CASE;
END IF;
IF UsesResources
  ChangeStatus(Hold);
  WAIT FOR PooledSpares[ResNumber] TO PriorityGive(SELF,NumResRequired,spcResPriority);
  END WAIT;
  PoolResAvailable[ResNumber]:=PooledSpares[ResNumber].Resources;
  IF EventsFile
  {if user selected event log from sim options dialog box}
    printString := REALTOSTR(SimTime) + "                ";
    printString := SUBSTR(1, 12, printString);
    outString:=printString;
    printString:= SUBSTR(1,20,BlockName + "                ");
    outString:=outString+printString;
    printString:= "  Obtained "+INTTOSTR(NumResRequired)+" Resource(s)";
    outString:=outString+printString;
    ASK EventStream TO WriteString(outString);
    {write a line indicating component is out of spares}
    ASK EventStream TO WriteLn;
  END IF;
  ChangeStatus(Repairing);
END IF;
IF (temp4 <= ProbAdjust)  AND (UseAdjust)
{Now we are using repair by adjustment.}
 IF (ARepairSeedNumber=9)
  ASK SysStreamA TO DrawNumber(ARepairDist,ARepairParam,TimeToARepair);
 ELSIF (ARepairSeedNumber=10)
  ASK SysStreamB TO DrawNumber(ARepairDist,ARepairParam,TimeToARepair);
 ELSE
  ASK ARepairStream[i] TO DrawNumber(ARepairDist,ARepairParam,TimeToARepair);
 END IF;
 {gets a repair time from the appropriate random number stream}
 IF (TimeToARepair<0.0)
  GetErrorBox("56: Negative repair time rounded to zero!");
  {displays an error message if time to repair is negative}
  TimeToARepair:=0.0;
  {set the time to repair to zero}
 END IF;
```

```
{waits until repair is completed.  This cannot be interrupted. Interrupt
statement only interrupts components that are green}
 WAIT DURATION TimeToARepair;
 END WAIT;
 BlockCost := BlockCost + TimeToARepair * RCost;
 {Update the appropriate block cost to reflect the repair time.}
 IF UsesResources
    ASK PooledSpares[ResNumber] TO TakeBack(SELF,NumResRequired);
    PoolResAvailable[ResNumber]:=PooledSpares[ResNumber].Resources;
 END IF;
 CompTimeToRepair:=TimeToARepair;
 CompTimeToARepair:=TimeToARepair;
ELSE
 IF (RepairSeedNumber=9)
  ASK SysStreamA TO DrawNumber(RepairDist,RepairParam,TimeToRepair);
 ELSIF (RepairSeedNumber=10)
  ASK SysStreamB TO DrawNumber(RepairDist,RepairParam,TimeToRepair);
 ELSE
  ASK RepairStream[i] TO DrawNumber(RepairDist,RepairParam,TimeToRepair);
 END IF;
{gets a repair time from the appropriate random number stream}
 IF (TimeToRepair<0.0)
  GetErrorBox("56: Negative repair time rounded to zero!");
  {displays an error message if time to repair is negative}
  TimeToRepair:=0.0;
  {set the time to repair to zero}
 END IF;
{waits until repair is completed.  This cannot be interrupted. Interrupt
statement only interrupts components that are green}
 WAIT DURATION TimeToRepair;
 END WAIT;
 BlockCost := BlockCost + TimeToRepair * RRCost;
 {Update the appropriate block cost for the repair time.}
 IF UsesResources
    ASK PooledSpares[ResNumber] TO TakeBack(SELF,NumResRequired);
    PoolResAvailable[ResNumber]:=PooledSpares[ResNumber].Resources;
 END IF;
 CompTimeToRepair:=TimeToRepair;
 CompTimeToRRepair:=TimeToRepair;
END IF;
{track time to repair for all components for use in MRT output}
ChangeStatus(Running);
BlockTimeToFail := SimTime-LastBlockColorChange;
BlockCost := BlockCost + BlockTimeToFail*DCost;
LastBlockColorChange := SimTime;
 {Updates the block cost to reflect the block down time.}
{changes component status to Green}
NumCurrentlyFailed:=NumCurrentlyFailed-1;
{decreased the counter tracking number of system components currently failed}
ReconfigureRepair;
{propagates the repair through the system}
IF (NumCurrentlyFailed=0)
{if no components in the entire system are failed}
  ASK System TO ChangeStatus(GGreen);
  {call a routine changing the system to green}
END IF;
IF EventsFile
{user selected event log}
  CASE System.Status
    WHEN GGreen:
      statusString:="Green";
    WHEN YYellow:
      statusString:="Yellow";
    WHEN RRed:
      statusString:="Red";
  END CASE;
```

```
        {converts system status to a string}
        printString := REALTOSTR(SimTime) + "                ";
        printString := SUBSTR(1, 12, printString);
        outString:=printString;
        printString:= SUBSTR(1,20,BlockName + "                ");
        outString:=outString+printString;
        printString:= "  Repaired,   RepairTime="+REALTOSTR(TimeToRepair);
        outString:=outString+printString;
        printString:= "  System="+statusString;
        outString:=outString+printString;
        ASK EventStream TO WriteString(outString);
        {writes a line in the event log indicating componet failure}
        ASK EventStream TO WriteLn;
      END IF;
      IF (SystemDependence=TRUE) AND (System.Status=RRed)
      {if the component is system dependent and the system is now down}
        ChangeStatus(Idle);
        WAIT FOR SystemTrigger TO Fire;
        {tells the component to wait until the system is up before operating}
        END WAIT;
        ChangeStatus(Running);
        LastBlockColorChange := SimTime;
      END IF;
     {  IF ((StreamDependence=1) AND (NOT(Trigger1Fired)))
        ChangeStatus(Idle);
        WAIT FOR Stream1Trigger TO Fire;
        END WAIT;
        ChangeStatus(Running);
        LastBlockColorChange := SimTime;
      END IF;
      IF (StreamDependence=2) AND NOT(Trigger2Fired)
        ChangeStatus(Idle);
        WAIT FOR Stream2Trigger TO Fire;
        END WAIT;
        ChangeStatus(Running);
        LastBlockColorChange := SimTime;
      END IF;
      IF (StreamDependence=3) AND NOT(Trigger3Fired)
        ChangeStatus(Idle);
        WAIT FOR Stream3Trigger TO Fire;
        END WAIT;
        ChangeStatus(Running);
        LastBlockColorChange := SimTime;
      END IF;}
    END IF;  {Status=Repairing}
    {if the component is not permanently failed (was repaired) it will reenter the loop}
  END WHILE;
END METHOD;    {Operate}
        ENTIRE OPERATE WITH COLD STANDBY REMOVED
        REPAIR FAILED UNIT INFORMATION
TELL METHOD RepairFailedUnit(IN i:INTEGER);
        51 LINES REMOVED
  CompTimeToRepair:=TimeToRepair;
  CompTimeToRRepair:=TimeToRepair;
  IF (Status=Repairing)
        45 LINES REMOVED
        CONDUCT EVENT INFORMATION
TELL METHOD ConductEvent();
        16 LINES REMOVED
  IF (draw=1.0)
    ChangeStatus(Success);
    BlockCost := BlockCost + 0.0;
  ELSE
    ChangeStatus(Failure);
    BlockCost := BlockCost + FCost;
    {Updates the block cost to reflect failed event block.}
```

```
      INC(NumCurrentlyFailed);
      ReconfigureFail;
      ReconfigureSystem;
    END IF;
END METHOD;  {ConductEvent}
          METHOD GENERATE SPARES REMOVED
          METHOD ORDER A SPARE REMOVED
          INITIALIZE INFORMATION
ASK METHOD Initialize(IN i:INTEGER; IN isBlock : BOOLEAN);
  VAR
          10 LINES REMOVED
      StreamDependence:= block.streamDepend;
      FailSeedNumber:=block.failStream;
      RepairSeedNumber:=block.repairStream;
      ARepairSeedNumber:=block.arepairStream;
      FailDist:=block.failDistro;
      NumFailDistParam:=block.numFailParams;
      RepairDist:=block.repairDistro;
      NumRepairDistParam:=block.numRepairParams;
      ARepairDist:=block.arepairDistro;
      NumARepairDistParam:=block.numARepairParams;
      NumDownStreamComp:=block.EFPAConnectOut;
      LogisticsDelay:=block.ALDTDelay;
      ALDTCost  := block.ALDTCostVal;
      SpareCost := block.SpareCostVal;
      UseAdjust:=block.useAdj;
      ProbAdjust:=block.probAdj;
      UCost := block.uCostFactor;
      DCost := block.dCostFactor;
      FCost := block.fCostFactor;
      ICost := block.iCostFactor;
      SCost := block.sCostFactor;
      LFactor := block.lFactor;
      RRAdjfactor:=block.rradjfactor;
      RRLimfactor:=block.rrlimfactor;
      RAdjfactor := block.radjfactor;
      RLimfactor := block.rlimfactor;
      RRCost    := block.rrcost;
      RCost     := block.rcost;
      InfiniteSpares:=block.infiniteSpares;
          16 LINES REMOVED
      IF SystemDependence
        ASK SysDepGroup TO Add(Block[i]);
      END IF;
      IF (StreamDependence=1)
        ASK Strm1DepGroup TO Add(Block[i]);
      END IF;
      IF (StreamDependence=2)
        ASK Strm2DepGroup TO Add(Block[i]);
      END IF;
      IF (StreamDependence=3)
        ASK Strm3DepGroup TO Add(Block[i]);
      END IF;
      IF (SparingMethod = Custom)
          10 LINES REMOVED
      NEW(ARepairParam, 1..NumARepairDistParam);
      FOR j:=1 TO NumARepairDistParam
        ARepairParam[j]:=block.arepairVals[j];
      END FOR;
      NEW(DownStreamBlock, 1..NumDownStreamComp);
          18 LINES REMOVED
      Gcost := node.ngcost;
      Ycost := node.nycost;
      Rcost := node.nrcost;
      IF (Type = Node)
          9 LINES REMOVED
```

```
            SET TO INIT VALUE INFORMATION
 ASK METHOD SetToInitValues(IN i:INTEGER);
   BEGIN
     SysStrm := 2;
     NumGoodPaths:=NumPathsIn;
     LastNodeColorChange := SimTime;
     NodeCost := 0.0;
     LastBlockColorChange := SimTime;
     BlockCost := 0.0;
     NumActivePaths:=NumPathsIn;
            12 LINES REMOVED
            RESET NODE STATISTICS INFORMATION
TELL METHOD ResetNodeStatistics();
 BEGIN
   LastNodeColorChange := SimTime;
   NodeCost := 0.0;
END METHOD;    {ResetNodeStatistics}
TELL METHOD ResetBlockStatistics();
 BEGIN
   LastBlockColorChange := SimTime;
   BlockCost := 0.0;
END METHOD;    {ResetBlockStatistics}
            TERMINATE INFORMATION
 ASK METHOD Terminate;
   BEGIN
     DISPOSE(FailDistParam);
     DISPOSE(RepairParam);
     DISPOSE(ARepairParam);
     DISPOSE(DownStreamBlock);
     {disposes of the arrays that contain distribution parameters and downstream block info}
 END METHOD;
            METHOD CHANGE STATUS REMOVED
END OBJECT;
            ALL SYSTEM OBJ CODE REMOVED
            FILEN INFORMATION
            23 LINES REMOVED
            PROCEDURE READ DATA REMOVED
            PROCEDURE NEW FILE REMOVED
            PROCEDURE CLOSE FILE REMOVED
            OPEN FILE INFORMATION
            27 LINES REMOVED
  newSpArr,
  emerTime,emerNumber,
  linkStartX,
            24 LINES REMOVED
  repVars,
  arepVars,
  realArray,holdArray, holdArray2    : valueArray;
  boolsArray   : boolArray;
            101 LINES REMOVED
    IF raptor30
      ASK saveFile TO ReadInt(totalPools);
      NEW(posArray,1..2);
      NEW(intArray, 1..14);
      NEW(realArray, 1..22);
      NEW(boolsArray, 1..4);
      NEW(stringArray, 1..3);
      FOR i := 1 TO totalBlocks   {go into a loop for all the blocks}
        FOR j:= 1 TO 14
          ASK saveFile TO ReadInt(nextInt);
          intArray[j] := nextInt;
        END FOR;
        FOR j := 1 TO 2
          ASK saveFile TO ReadReal(nextReal);
          posArray[j] := nextReal;
        END FOR;
```

```
FOR j := 1 TO 6
  ASK saveFile TO ReadReal(nextReal);
  realArray[j] := nextReal;
END FOR;
ASK saveFile TO ReadReal(nextReal);
    10 LINES REMOVED
realArray[13] := nextReal;
ASK saveFile TO ReadReal(nextReal);
realArray[14] := nextReal;
ASK saveFile TO ReadReal(nextReal);
realArray[15] := nextReal;
ASK saveFile TO ReadReal(nextReal);
realArray[16] := nextReal;
ASK saveFile TO ReadReal(nextReal);
realArray[17] := nextReal;
ASK saveFile TO ReadReal(nextReal);
realArray[18] := nextReal;
ASK saveFile TO ReadReal(nextReal);
realArray[19] := nextReal;
ASK saveFile TO ReadReal(nextReal);
realArray[20] := nextReal;
ASK saveFile TO ReadReal(nextReal);
realArray[21] := nextReal;
ASK saveFile TO ReadReal(nextReal);
realArray[22] := nextReal;
{Array elements 13-19 were added later to handle additional features,
including repair versus remove and replace, reliability growth/degradation,
and cost calculations.}
FOR j := 1 TO 4
  ASK saveFile TO ReadInt(nextInt);
  IF nextInt = 1
    boolsArray[j] := TRUE;
  ELSE
    boolsArray[j] := FALSE;
  END IF;
END FOR;
FOR j := 1 TO 3
    22 LINES REMOVED
ASK saveFile TO ReadReal(nextReal);
  repVars[j] := nextReal;
END FOR;
NEW(arepVars, 1..intArray[12]);
FOR j := 1 TO intArray[12]
  ASK saveFile TO ReadReal(nextReal);
  arepVars[j] := nextReal;
END FOR;
NEW(block);
    8 LINES REMOVED
ASK block TO Draw;
NEW(holdArray, 1..7);
holdArray[1] := realArray[1];
holdArray[2] := realArray[2];
holdArray[3] := realArray[3];
holdArray[4] := realArray[4];
holdArray[5] := realArray[5];
holdArray[6] := realArray[18];
holdArray[7] := realArray[19];
ASK block TO SetSpecialData(holdArray, boolsArray, stringArray[2],
                    stringArray[3], intArray[10],intArray[14],
                    sparing);
DISPOSE(holdArray);
NEW(holdArray, 1..6);
holdArray[1] := realArray[9];
holdArray[2] := realArray[10];
holdArray[3] := realArray[11];
holdArray[4] := realArray[12];
```

```
        holdArray[5] := realArray[16];
        holdArray[6] := realArray[17];
        NEW(holdArray2, 1..7);
        holdArray2[1] := realArray[7];
        holdArray2[2] := realArray[13];
        holdArray2[3] := realArray[14];
        holdArray2[4] := realArray[15];
        holdArray2[5] := realArray[20];
        holdArray2[6] := realArray[21];
        holdArray2[7] := realArray[22];
        ASK block TO SetBlockData(stringArray[1], intArray[5], intArray[6],
                        intArray[3], intArray[7], intArray[8],
                        intArray[4], intArray[11], intArray[12],
                        intArray[13],failVars, repVars, arepVars, holdArray,
                        holdArray2, boolsArray[4]);
        DISPOSE(holdArray);
        DISPOSE(holdArray2);
{embeds the data into the new block}
            58 LINES REMOVED
        IF node.connectIntoNum > 1
          ASK node TO SetKofN(node.goodPaths, node.connectIntoNum);
        END IF;
        NEW (realArray,1..3);
        ASK saveFile TO ReadReal(nextReal);
        realArray[1] := nextReal;
        ASK saveFile TO ReadReal(nextReal);
        realArray[2] := nextReal;
        ASK saveFile TO ReadReal(nextReal);
        realArray[3] := nextReal;
        ASK node TO SetNodeCost(realArray[1], realArray[2], realArray[3]);
        DISPOSE(realArray);
      END FOR;
      NEW(intArray,1..4);
      FOR i := 1 TO totalLinks
            73 LINES REMOVED
        ASK saveFile TO ReadReal(emerTime);
        ASK saveFile TO ReadReal(emerNumber);
        ASK saveFile TO ReadInt(nextInt);
        IF nextInt=1
          emerFlag := TRUE;
        ELSE
          emerFlag := FALSE;
        END IF;
        NEW(pool);
        ASK pool TO SetData(poolName, sparing, initSp, newSp, newSpArr, emerTime,
                    emerNumber, emerFlag);
      END FOR;
    ELSE
      NEW(message, 1..3);
            86 LINES REMOVED
        realArray[6] := oldRec.pSuccess;
        realArray[7] := oldRec.probAdjust;
        realArray[9] := oldRec.RRAdjFact;
        realArray[10] := oldRec.RRLimFact;
        realArray[11] := oldRec.RAdjFact;
        realArray[12] := oldRec.RLimFact;
        realArray[13] := oldRec.uCostFactor;
        realArray[14] := oldRec.dCostFactor;
        realArray[15] := oldRec.fCostFactor;
        realArray[16] := oldRec.RRCostFact;
        realArray[17] := oldRec.RCostFact;
        realArray[18] := oldRec.ALDTCostVal;
        realArray[19] := oldRec.SpareCostVal;
        realArray[20] := oldRec.iCostFactor;
        realArray[21] := oldRec.sCostFactor;
        realArray[22] := oldRec.lFactor;
```

```
        NEW(boolsArray, 1..4);
        boolsArray[1] := oldRec.depends;
        boolsArray[2] := oldRec.infiniteSpares;
        boolsArray[3] := FALSE;
        boolsArray[4] := oldRec.useAdjust;
        NEW(holdArray, 1..7);
        holdArray[1] := realArray[1];
        holdArray[2] := realArray[2];
        holdArray[3] := realArray[3];
        holdArray[4] := realArray[4];
        holdArray[5] := realArray[5];
        holdArray[6] := realArray[18];
        holdArray[7] := realArray[19];
        IF oldRec.infiniteSpares
          ASK block TO SetSpecialData(holdArray, boolsArray, "unnamed", "unnamed", 0,0, Infinite);
        ELSIF ((realArray[1]=0.)AND((realArray[5]=0.)OR(realArray[2]=0.))AND(realArray[3]=0.))
          ASK block TO SetSpecialData(holdArray, boolsArray, "unnamed", "unnamed", 0,0, None);
        ELSE
          ASK block TO SetSpecialData(holdArray, boolsArray, "unnamed", "unnamed", 0,0, Custom);
        END IF;
        DISPOSE(holdArray);
        newBlockName := CHARTOSTR(oldRec.name);
        FOR j := 1 TO STRLEN(newBlockName)
          IF SUBSTR(j, j, newBlockName) = " "
            REPLACE(newBlockName, j, j, "_");
          END IF;
        END FOR;
        NEW(holdArray, 1..6);
        holdArray[1] := oldRec.RRAdjFact;
        holdArray[2] := oldRec.RRLimFact;
        holdArray[3] := oldRec.RAdjFact;
        holdArray[4] := oldRec.RLimFact;
        holdArray[5] := oldRec.RRCostFact;
        holdArray[6] := oldRec.RCostFact;
        NEW(holdArray2, 1..7);
        holdArray2[1] := oldRec.probAdjust;
        holdArray2[2] := oldRec.uCostFactor;
        holdArray2[3] := oldRec.dCostFactor;
        holdArray2[4] := oldRec.fCostFactor;
        holdArray2[5] := oldRec.iCostFactor;
        holdArray2[6] := oldRec.sCostFactor;
        holdArray2[7] := oldRec.lFactor;
        ASK block TO SetBlockData(newBlockName, oldRec.failDist,
                        oldRec.numFailParams, oldRec.failSeed,
                        oldRec.repDist, oldRec.numRepParams,
                        oldRec.repSeed, oldRec.arepDist, oldRec.numARepParams,
                        oldRec.arepSeed, failVars, repVars, arepVars, holdArray,
                        holdArray2, oldRec.useAdjust);
        DISPOSE(holdArray);
        DISPOSE(holdArray2);
{embeds the data into the new block}
            44 LINES REMOVED
        ASK node TO SetKofN(node.goodPaths, node.connectIntoNum);
        END IF;
        NEW(realArray, 1..3);
        realArray[1] := oldRec.oldgcost;
        realArray[2] := oldRec.oldycost;
        realArray[3] := oldRec.oldrcost;
        ASK node TO SetNodeCost(realArray[1], realArray[2],realArray[3]);
        DISPOSE(realArray);
      END FOR;
      FOR i := 1 TO totalLinks
            73 LINES REMOVED
            SAVE FILE INFORMATION
            40 LINES REMOVED
```

```
    ASK saveFile TO WriteInt(block.connectOutOfNum, 6);
    ASK saveFile TO WriteInt(block.numResources, 6);
    ASK saveFile TO WriteInt(block.arepairDistro, 6);
    ASK saveFile TO WriteInt(block.numARepairParams, 6);
    ASK saveFile TO WriteInt(block.arepairStream, 6);
    ASK saveFile TO WriteInt(block.streamDepend, 6);
    ASK saveFile TO WriteReal(block.xPosition, 15, 6);
          6 LINES REMOVED
    ASK saveFile TO WriteReal(block.pSuccess, 20, 6);
    ASK saveFile TO WriteReal(block.probAdj, 20, 6);
    ASK saveFile TO WriteReal(block.rradjfactor, 20, 6);
    ASK saveFile TO WriteReal(block.rrlimfactor, 20, 6);
    ASK saveFile TO WriteReal(block.radjfactor, 20, 6);
    ASK saveFile TO WriteReal(block.rlimfactor, 20, 6);
    ASK saveFile TO WriteReal(block.uCostFactor, 20, 6);
    ASK saveFile TO WriteReal(block.dCostFactor, 20, 6);
    ASK saveFile TO WriteReal(block.fCostFactor, 20, 6);
    ASK saveFile TO WriteReal(block.rrcost, 20, 6);
    ASK saveFile TO WriteReal(block.rcost, 20, 6);
    ASK saveFile TO WriteReal(block.ALDTCostVal, 20, 6);
    ASK saveFile TO WriteReal(block.SpareCostVal, 20, 6);
    ASK saveFile TO WriteReal(block.iCostFactor, 20, 6);
    ASK saveFile TO WriteReal(block.sCostFactor, 20, 6);
    ASK saveFile TO WriteReal(block.lFactor, 20, 6);
    {Additional features were added to handle repair by adjustment as well as remove and
    replace, along with reliability growth/degradation, along with calculating an
    overall maintenance and operations cost.}
    IF block.systemDepend
          13 LINES REMOVED
    END IF;
    IF block.useAdj
      ASK saveFile TO WriteInt(1, 3);
    ELSE
      ASK saveFile TO WriteInt(0, 3);
    END IF;
    ASK saveFile TO WriteString(" "+block.blockName+" ");
          17 LINES REMOVED
      ASK saveFile TO WriteReal(block.repairVals[j], 20, 6);
    END FOR;
    FOR j := 1 TO block.numARepairParams
      ASK saveFile TO WriteReal(block.arepairVals[j], 20, 6);
    END FOR;
    ASK saveFile TO WriteLn;
          9 LINES REMOVED
    ASK saveFile TO WriteInt(node.goodPaths, 6);
    ASK saveFile TO WriteReal(node.ngcost, 15, 6);
    ASK saveFile TO WriteReal(node.nycost, 15, 6);
    ASK saveFile TO WriteReal(node.nrcost, 15, 6);
    ASK saveFile TO WriteLn;
  END FOR;
  FOR i := 1 TO totalLinks
          33 LINES REMOVED
    ASK saveFile TO WriteReal(pool.emergencyTime, 20, 6);
    ASK saveFile TO WriteLn;
    ASK saveFile TO WriteReal(pool.emergencyNumber, 20, 6);
    ASK saveFile TO WriteLn;
    IF pool.emergencyFlag
          21 LINES REMOVED
          PROCEDURE SAVE AS A FILE REMOVED
          PROCEDURE ASK FOR A SAVE REMOVED
          PROCEDURE GET FILE NAME REMOVED
END MODULE. {FileN}
          RBD BLOC INFORMATION
          15 LINES REMOVED
          PROCEDURE ARROWHEAD REMOVED
          ALL RBD BASIC OBJ REMOVED
```

```
            RBD NODE OBJ INFORMATION
OBJECT RBDNodeObj;
            METHOD SET TYPE REMOVED
            METHOD SET GOOD PATHS REMOVED
            METHOD SET NUM REMOVED
            METHOD SET K OUT OF N REMOVED
{-------------------------------------------------------------------------------------------}
{                                                                                           }
{                          ASK METHOD SetNodeCost                                           }
{                                                                                           }
{ Sets the node's cost.                                                                     }
{                                                                                           }
{-------------------------------------------------------------------------------------------}
  ASK METHOD SetNodeCost(IN costg, costy, costr : REAL);
  BEGIN
    ngcost := costg;
    nycost := costy;
    nrcost := costr;
  END METHOD;
END OBJECT; {RBDNodeObj}
            ALL OF LINK OBJECT REMOVED
            RBD BLOCK OBJ INFORMATION
OBJECT RBDBlockObj;
            SET BLOCK DATA INFORMATION
  ASK METHOD SetBlockData(IN bName                    : STRING;
                  IN fDist, numFParms, fStream,
                    rDist, numRParms, rStream,
                    arDist, numARParms, arStream    : INTEGER;
                  IN fVals, rVals, arVals, rfactvals      : valueArray;
                  IN holdvals  :  valueArray;
                  IN uAdj  :  BOOLEAN);
            9 LINES REMOVED
    DISPOSE(repairVals);
    DISPOSE(arepairVals);
    blockName := bName;
            8 LINES REMOVED
    numRepairParams := numRParms;
    repairStream := rStream;
    arepairDistro := arDist;
    numARepairParams := numARParms;
    arepairStream  := arStream;
    probAdj       := holdvals[1];
    rradjfactor     := rfactvals[1];
    rrlimfactor     := rfactvals[2];
    radjfactor      := rfactvals[3];
    rlimfactor      := rfactvals[4];
    rrcost        := rfactvals[5];
    rcost         := rfactvals[6];
    useAdj        := uAdj;
    uCostFactor    := holdvals[2];
    dCostFactor    := holdvals[3];
    fCostFactor    := holdvals[4];
    iCostFactor    := holdvals[5];
    sCostFactor    := holdvals[6];
    lFactor       := holdvals[7];
    {Additional parameters track cost features, reliability growth/degradation, and allowing
    the user to repair by adjust only as well as remove and replace.}
    IF failDistro = 17
            8 LINES REMOVED
    NEW(repairVals, 1..numRParms);
    NEW(arepairVals, 1..numARParms);
    FOR i := 1 TO numFParms
      failVals[i] := fVals[i];
    END FOR;
    FOR i := 1 TO numRParms
      repairVals[i] := rVals[i];
```

```
  END FOR;
  FOR i := 1 TO numARParms
    arepairVals[i] := arVals[i];
  END FOR;
END METHOD; {SetBlockData}
        SET SPECIAL DATA INFORMATION
ASK METHOD SetSpecialData(IN realArray        : valueArray;
                  IN boolsArray       : boolArray;
                  IN bPoolName, bResName : STRING;
                  IN bNumRes, streamdepend        : INTEGER;
                  IN bSpareType       : SparingType);
BEGIN
  {set internal variables for special repair data equal to the values passed in}
  numSpares     := TRUNC(realArray[1]);
  spareRate     := realArray[2];
  spareDelay    := realArray[3];
  ALDTDelay     := realArray[4];
  numPerVal     := realArray[5];
  ALDTCostVal   := realArray[6];
  SpareCostVal  := realArray[7];
  systemDepend  := boolsArray[1];
  infiniteSpares := boolsArray[2];
  useResource   := boolsArray[3];
  streamDepend  := streamdepend;
  IF useResource
        8 LINES REMOVED
        DISPLAY DETAILS INFORMATION
ASK METHOD DisplayDetails;
        8 LINES REMOVED
  spareArray, holdArray, holdArray2 : valueArray;
  boolsArray : boolArray;
        8 LINES REMOVED
  spRate := spareRate;
  NEW(numArray, 1..11);
  numArray[1] := failDistro;
  numArray[2] := numFailParams;
  numArray[3] := failStream;
  numArray[4] := repairDistro;
  numArray[5] := numRepairParams;
  numArray[6] := repairStream;
  numArray[8] := arepairDistro;
  numArray[9] := numARepairParams;
  numArray[10] := arepairStream;
  numArray[7] := numResources;
  numArray[11] := streamDepend;
  NEW(spareArray, 1..22);
  spareArray[1] := nSpares;
  spareArray[2] := spRate;
  spareArray[3] := spareDelay;
  spareArray[4] := ALDTDelay;
  spareArray[5] := numPerVal;
  spareArray[6] := pSuccess;
  spareArray[7] := probAdj;
  spareArray[9] := rradjfactor;
  spareArray[10] := rrlimfactor;
  spareArray[11] := radjfactor;
  spareArray[12] := rlimfactor;
  spareArray[13] := uCostFactor;
  spareArray[14] := dCostFactor;
  spareArray[15] := fCostFactor;
  spareArray[16] := rrcost;
  spareArray[17] := rcost;
  spareArray[18] := ALDTCostVal;
  spareArray[19] := SpareCostVal;
  spareArray[20] := iCostFactor;
  spareArray[21] := sCostFactor;
```

```
    spareArray[22] := lFactor;
    NEW(boolsArray, 1..5);
    boolsArray[1] := systemDepend;
    boolsArray[2] := infiniteSpares;
    boolsArray[3] := useResource;
    boolsArray[4] := operatingBlock;
    boolsArray[5] := useAdj;
    noRemain := TRUE;
    ASK propBox TO GetProperties(blockName, numArray, failVals, repairVals, arepairVals, spareArray,
                    boolsArray,poolName, resourceName, sparingType);
{call the propBox's GetProperties method (see IDialogs)}
    pSuccess      := spareArray[6];
    failDistro    := numArray[1];
    numFailParams  := numArray[2];
    failStream    := numArray[3];
    repairDistro   := numArray[4];
    numRepairParams := numArray[5];
    repairStream   := numArray[6];
    arepairDistro  := numArray[8];
    numARepairParams:= numArray[9];
    arepairStream  := numArray[10];
    numResources   := numArray[7];
    probAdj       := spareArray[7];
    useAdj        := boolsArray[5];
    rradjfactor     := spareArray[9];
    rrlimfactor     := spareArray[10];
    radjfactor     := spareArray[11];
    rlimfactor     := spareArray[12];
    uCostFactor    := spareArray[13];
    dCostFactor    := spareArray[14];
    fCostFactor    := spareArray[15];
    rrcost        := spareArray[16];
    rcost         := spareArray[17];
    ALDTCostVal    := spareArray[18];
    SpareCostVal   := spareArray[19];
    iCostFactor    := spareArray[20];
    sCostFactor    := spareArray[21];
    lFactor       := spareArray[22];
    NEW(holdArray, 1..7);
    holdArray[1] := spareArray[1];
    holdArray[2] := spareArray[2];
    holdArray[3] := spareArray[3];
    holdArray[4] := spareArray[4];
    holdArray[5] := spareArray[5];
    holdArray[6] := spareArray[18];
    holdArray[7] := spareArray[19];
    SetSpecialData(holdArray, boolsArray, poolName, resourceName,
            numResources, numArray[11],sparingType);
{reset the blocks internal variables with the new data}
    DISPOSE(holdArray);
    blockImage := Child("BasicBlock", 601);
    IF NOT boolsArray[4]
      DISPOSE(failVals);
      DISPOSE(repairVals);
      DISPOSE(arepairVals);
      NEW(failVals,1..1);
      NEW(repairVals,1..1);
      NEW(arepairVals,1..1);
      failVals[1] := pSuccess;
      repairVals[1] := pSuccess;
      arepairVals[1] := pSuccess;
      NEW(holdArray, 1..6);
      holdArray[1] := 1.0;
      holdArray[2] := 2.0;
      holdArray[3] := 1.0;
      holdArray[4] := 2.0;
```

```
  holdArray[5] := 0.0;
  holdArray[6] := 0.0;
  NEW(holdArray2, 1..7);
  holdArray2[1] := probAdj;
  holdArray2[2] := uCostFactor;
  holdArray2[3] := dCostFactor;
  holdArray2[4] := fCostFactor;
  holdArray2[5] := iCostFactor;
  holdArray2[6] := sCostFactor;
  holdArray2[7] := lFactor;
  SetBlockData(blockName, 17, 1, failStream, 17, 1, failStream,
        17, 1, failStream, failVals, repairVals, arepairVals,
        holdArray,holdArray2, FALSE);
  rradjfactor := holdArray[1];
  rrlimfactor := holdArray[2];
  radjfactor := holdArray[3];
  rlimfactor := holdArray[4];
  rrcost    := holdArray[5];
  rcost     := holdArray[6];
  probAdj := holdArray2[1];
  uCostFactor := holdArray2[2];
  dCostFactor := holdArray2[3];
  fCostFactor := holdArray2[4];
  iCostFactor := holdArray2[5];
  sCostFactor := holdArray2[6];
  lFactor    := holdArray2[7];
  DISPOSE(holdArray);
  DISPOSE(holdArray2);
  ASK blockImage TO SetColor(Cyan);
ELSE
  NEW(holdArray, 1..6);
  holdArray[1] := rradjfactor;
  holdArray[2] := rrlimfactor;
  holdArray[3] := radjfactor;
  holdArray[4] := rlimfactor;
  holdArray[5] := rrcost;
  holdArray[6] := rcost;
  NEW(holdArray2, 1..7);
  holdArray2[1] := probAdj;
  holdArray2[2] := uCostFactor;
  holdArray2[3] := dCostFactor;
  holdArray2[4] := fCostFactor;
  holdArray2[5] := iCostFactor;
  holdArray2[6] := sCostFactor;
  holdArray2[7] := lFactor;
  SetBlockData(blockName, failDistro, numFailParams,
        failStream, repairDistro, numRepairParams, repairStream,
        arepairDistro, numARepairParams, arepairStream,
        failVals, repairVals, arepairVals, holdArray, holdArray2, useAdj);
  rradjfactor := holdArray[1];
  rrlimfactor := holdArray[2];
  radjfactor := holdArray[3];
  rlimfactor := holdArray[4];
  rrcost    := holdArray[5];
  rcost     := holdArray[6];
  probAdj := holdArray2[1];
  uCostFactor := holdArray2[2];
  dCostFactor := holdArray2[3];
  fCostFactor := holdArray2[4];
  iCostFactor := holdArray2[5];
  sCostFactor := holdArray2[6];
  lFactor    := holdArray2[7];
  DISPOSE(holdArray);
  DISPOSE(holdArray2);
  ASK blockImage TO SetColor(Green);
```

```
            8 LINES REMOVED
            METHOD SET CONNECT TO NODE REMOVED
            METHOD COPY REST OF DATA REMOVED
            METHOD SET POOL NUMBER REMOVED
            METHOD SET SPARING INFINITE REMOVED
            METHOD SET NO RESOURCES REMOVED
            RBD RES INFORMATION
            16 LINES REMOVED
            SHOW POOLS INFORMATION
PROCEDURE ShowPools;
VAR
    value2, value3,
    value4, value5        : ValueBoxObj;
    combo         : ComboBoxObj;
    check1        : CheckBoxObj;
            10 LINES REMOVED
    value4 := ASK poolBox Child("Value4", 558);
    value5 := ASK poolBox Child("Value5", 568);
    combo  := ASK poolBox Child("ComboBox", 555);
    check1  := ASK poolBox Child("EmerFlagBox", 583);
    label2 := ASK poolBox Child("ValLabel2", 560);
            8 LINES REMOVED
    ASK value4 TO Deactivate;
    ASK value5 TO Deactivate;
    ASK label2 TO Deactivate;     {Deactivate the arrival rate and emergency spares stuff}
            11 LINES REMOVED
            PROCEDURE GET POOLS LIST REMOVED
            SPARES POOL OBJ INFORMATION
OBJECT SparePoolObj;
            SET DATA INFORMATION
ASK METHOD SetData(IN name                : STRING;
              IN type                 : SparingType;
              IN poolNum, numSpares       : INTEGER;
              IN numSpareArrival, emerTime, emerNumber : REAL;
              IN emerFlag : BOOLEAN);
  BEGIN
            12 LINES REMOVED
    emergencyTime    := emerTime;
    emergencyNumber  := emerNumber;
    emergencyFlag := emerFlag;
    ASK poolGroup TO Add(SELF);
    {somethingChanged := TRUE;  }
  END METHOD;{SetData}
            METHOD REMOVE POOL REMOVED
END OBJECT; {SparePoolObj}
END MODULE; {IRBDRes}
            RBD WIN INFORMATION
            60 LINES REMOVED
{default number of repair distribution parameters}
  dRepStreamNum,
  dARepDistType,
  dARepDistParms,
  dARepStreamNum,
{default random stream number for the repair distribution}
  buttonNum,
{Id of a generic button}
  fromBlockId,
{Id of the block which a link connects from (out of)}
  imageId,
{Id of the selected system status image's radio button}
  blueObjId,
  dStrmDepen    : INTEGER;
{Id of the currently selected (highlighted blue) object}
            50 LINES REMOVED
{default start statistics gathering at failure # criteria}
  dProbSuccess,
```

```
  dUCost,
  dSCost,
  dLevel,
  dDCost,
  dFCost,
  dICost,
  dprobAdj,
  drradjfactor,
  drrlimfactor,
  dradjfactor,
  drlimfactor,
  drrcost,
  drcost,
  dALDTCost,
  dSpareCost  : REAL;
{default R value for an event block}
            27 LINES REMOVED
  freezing,
  duseAdj    : BOOLEAN;
{tells me if I'm in the process of freezing a RBD}
            12 LINES REMOVED
  radioBox     : RadioBoxObj;
  grncostBox, ylwcostBox, redcostBox : ValueBoxObj;
{a generic set of radio buttons}
  linkMessage  : ImageObj;
{the image of the message 'Click right mouse button to stop adding links'}
  linkText     : TextObj;
{the text of the above object}
  dFailParmVals,
  dRepParmVals,
  dARepParmVals : valueArray;
{arrays which hold the default values of failure and repair distribution parameters}
            16 LINES REMOVED
{default sparing type}
  realArray,realArray2    : valueArray;
{array of reals}
    boolsArray    : boolArray;
{array of booleans}
  message      : TextBufferType;
  result       : BOOLEAN;
{array of strings to be displayed using SendAlert procedure}
            INIT DISPLAY INFORMATION
            112 LINES REMOVED
    dSysDepen         := FALSE;
    dStrmDepen        := 0;
    dInfinSpares      := TRUE;
            10 LINES REMOVED
    dRepStreamNum     := 9;
    dARepDistType     := 7;
    dARepDistParms    := 2;
    dARepStreamNum    := 9;
    NEW(dFailParmVals, 1..dFailDistParms);
    dFailParmVals[1]  := 1.;
    dFailParmVals[2]  := 0.;
    NEW(dRepParmVals, 1..dRepDistParms);
    dRepParmVals[1] := 1.;
    dRepParmVals[2] := 1.;
    NEW(dARepParmVals, 1..dARepDistParms);
    dARepParmVals[1] := 1.;
    dARepParmVals[2] := 1.;
    dUseResources     := FALSE;
    dOperatingBlock   := TRUE;
    dProbSuccess      := 0.8;
    dUCost := 0.0;
    dSCost := 0.0;
    dDCost := 0.0;
```

```
    dFCost := 0.0;
    dICost := 0.0;
    dLevel := 1.0;
    drradjfactor      := 1.0;
    drrlimfactor      := 2.0;
    dradjfactor     := 1.0;
    drlimfactor     := 2.0;
    dNumRes         := 1;
    dBlockName      := "unnamed";
    dResName        := "unnamed";
    dPoolName       := "unnamed";
    dSpareType      := Infinite;
    dprobAdj        := 0.00;
    duseAdj         := FALSE;
    drrcost         := 0.0;
    drcost          := 0.0;
    dALDTCost       := 0.0;
    dSpareCost      := 0.0;
  END IF;
  menuitem := ASK menubar Descendant("separator", 0);
          49 LINES REMOVED
          PROCEDURE ADD BLOCK REMOVED
          PROCEDURE ADD CONNECTOR REMOVED
          PROCEDURE ADD NODE REMOVED
          PROCEDURE CLEAR ALL BLOCKS REMOVED
          GET BLOCK NAME INFORMATION
PROCEDURE GetBlockName(INOUT nameString                    : STRING;
              INOUT cancelledFlag, remain, sysDepen, operating : BOOLEAN;
              INOUT failStreamNum, repairStreamNum, arepairStreamNum,
                  strmDepen                    : INTEGER;
              INOUT Ps,Ucost, Scost, Dcost, Fcost, Icost, Level: REAL);
          6 LINES REMOVED
  checkBox      : CheckBoxObj;
  streamRadBox,
  strmdepRadBox,
  typeRadBox    : RadioBoxObj;
  streamRadButton,
  typeRadButton,
  strmdepRadButton : RadioButtonObj;
  rVal, uVal, dVal, fVal, iVal, sVal, lVal        : ValueBoxObj;
  dialogBox     : DetailsBoxObj;
          7 LINES REMOVED
{set the checkbox to the dialog box's check box}
  streamRadBox := ASK dialogBox Child("StreamRadioBox", 918);
  typeRadBox := ASK dialogBox Child("TypeRadioBox", 919);
{get a handle to the radio buttons as well}
  rVal := ASK dialogBox Child("RValBox", 920);
  ASK rVal TO SetValue(Ps);
  uVal := ASK dialogBox Child("UCostBox", 923);
  ASK uVal TO SetValue(Ucost);
  dVal := ASK dialogBox Child("DCostBox", 922);
  ASK dVal TO SetValue(Dcost);
  fVal := ASK dialogBox Child("FCostBox", 921);
  ASK fVal TO SetValue(Fcost);
  iVal := ASK dialogBox Child("ICostBox", 924);
  ASK iVal TO SetValue(Icost);
  sVal := ASK dialogBox Child("SCostBox", 925);
  ASK sVal TO SetValue(Scost);
  lVal := ASK dialogBox Child("LevelBox", 926);
  ASK lVal TO SetValue(Level);
  ASK text1 TO SetText(nameString);
{ask the text box to set its contents equal to the name string that was passed into this}
{procedure}
  ASK checkBox TO SetCheck(sysDepen);
{asked the check box to set its condition equal to the one passed in}
  strmdepRadBox := ASK dialogBox Child("StrmDepRadioBox",970);
```

```
  IF strmDepen = 0
    strmdepRadButton := ASK strmdepRadBox Child("NotRadButton",9282);
  ELSIF strmDepen = 1
    strmdepRadButton := ASK strmdepRadBox Child("Stream1RadButton",9283);
  ELSIF strmDepen = 2
    strmdepRadButton := ASK strmdepRadBox Child("Stream2RadButton",9284);
  ELSE
    strmdepRadButton := ASK strmdepRadBox Child("Stream3RadButton",9285);
  END IF;
{asked the combo box to set its condition equal to the one passed in}
  IF failStreamNum = 9
    streamRadButton := ASK streamRadBox Child("StreamARadButton", 9181);
  ELSIF failStreamNum = 10
    streamRadButton := ASK streamRadBox Child("StreamBRadButton", 9182);
  ELSE
    streamRadButton := ASK streamRadBox Child("IndepRadButton", 9183);
  END IF;
{this IF statement gets a handle to the radio button which should be currently selected}
  IF operating
{If an operating block select the operating button and disable R value}
    typeRadButton := ASK typeRadBox Child("OpRadButton", 9191);
    ASK typeRadBox TO SetSelectedButton(typeRadButton);
    ASK rVal TO SetSelectable(FALSE);
    ASK fVal TO SetSelectable(FALSE);
    ASK uVal TO SetSelectable(TRUE);
    ASK dVal TO SetSelectable(TRUE);
    ASK iVal TO SetSelectable(TRUE);
    ASK sVal TO SetSelectable(TRUE);
  ELSE
{If an event block select the event button and enable R value}
    typeRadButton := ASK typeRadBox Child("EventRadButton", 9192);
    ASK typeRadBox TO SetSelectedButton(typeRadButton);
    ASK rVal TO SetSelectable(TRUE);
    ASK fVal TO SetSelectable(TRUE);
    ASK uVal TO SetSelectable(FALSE);
    ASK dVal TO SetSelectable(FALSE);
    ASK iVal TO SetSelectable(TRUE);
    ASK sVal TO SetSelectable(FALSE);
  END IF;
{display the correct stream button}
  ASK streamRadBox TO SetSelectedButton(streamRadButton);
  ASK strmdepRadBox TO SetSelectedButton(strmdepRadButton);
  ASK dialogBox TO Draw;
            40 LINES REMOVED
      Ps := 0.8;
       {Ucost := uVal.Value();
       Dcost := dVal.Value();}
      ELSE
{if an event block, check for a valid R value}
       operating := FALSE;
       Ps := rVal.Value();
       {Fcost := fVal.Value();}
       IF ((Ps < 0.0) OR (Ps > 1.0))
         NEW(message, 1..1);
            16 LINES REMOVED
     END IF;
     IF strmdepRadBox.SelectedButton.Id = 9282
       strmDepen := 0;
     ELSIF strmdepRadBox.SelectedButton.Id = 9283
       strmDepen := 1;
     ELSIF strmdepRadBox.SelectedButton.Id  = 9284
       strmDepen := 2;
     ELSE
       strmDepen := 3;
     END IF;
   ELSIF ASK button ReferenceName = "CancelButton"
```

```
{if the 'cancel' button was clicked, set a flag for later use}
      cancelledFlag := TRUE;
      validData := TRUE;
    END IF;
  UNTIL validData;
  Ucost := uVal.Value();
  Dcost := dVal.Value();
  Fcost := fVal.Value();
  Icost := iVal.Value();
  Scost := sVal.Value();
  Level := lVal.Value();
  DISPOSE(dialogBox);
  IF indepFlag
{if independent streams were chosen, call the routine which gets the independent stream numbers}
    GetStream(("Failure Stream"), failStreamNum);
    IF operating
      GetStream(("Repair Stream"), repairStreamNum);
      arepairStreamNum := repairStreamNum;
    END IF;
  END IF;
END PROCEDURE; {GetBlockName}
          INIT GET DISTRO INFORMATION
PROCEDURE InitGetDistro(INOUT blockName                          : STRING;
              INOUT fDistroType, fNumParms, rDistroType, rNumParms,
                  arDistroType, arNumParms                 : INTEGER;
              INOUT fParmVals, rParmVals, arParmVals, rFactorVals      : valueArray;
              INOUT prAdjustment  : REAL;
              INOUT usAdjustment  : BOOLEAN);
          8 LINES REMOVED
{call the box's GetDistro method in IDialogs to handle input, and return values}
  ASK distrobox TO GetDistro(fDistroType, fNumParms, rDistroType, rNumParms,
                  arDistroType, arNumParms, fParmVals, rParmVals, arParmVals,
                  rFactorVals, prAdjustment, usAdjustment);
  DISPOSE(distrobox);
END PROCEDURE; {InitGetDistro}
          PROCEDURE GET STREAM REMOVED
          GET SPECIAL INFORMATION
PROCEDURE GetSpecial(IN barLbl                                   : STRING;
            INOUT sparesVal, sparesReplenVal, sparesDelayVal, ALDTVal,
                ALDTcost, Sparecost, numPerVal                : REAL;
            INOUT infinSpares, resources                    : BOOLEAN;
            INOUT poolName, resName                          : STRING;
            INOUT numRes                                   : INTEGER;
            INOUT spareType                                : SparingType);
VAR
  done,
  dummy       : BOOLEAN;
  specialBox    : SpecialBoxObj;
  button      : ButtonObj;
  value1, value2,
  value3, value4,
  value5, value6, value7, value8 : ValueBoxObj;
  label1, label2,
  label3      : LabelObj;
  newCheck,
  emerCheck,
  ALDTCheck,
  resCheck     : CheckBoxObj;
  spareCombo,
  coldCombo,
  resCombo     : ComboBoxObj;
  spareBox     : RadioBoxObj;
  spareButton   : RadioButtonObj;
  pool        : SparePoolObj;
          15 LINES REMOVED
 value6 := ASK specialBox Child("ResourceRequired", 810);
```

```
  value7 := ASK specialBox Child("ALDTCostBox", 851);
  value8 := ASK specialBox Child("SpareCostBox", 850);
  newCheck :=  ASK specialBox Child("NewCheckBox", 812);
  emerCheck := ASK specialBox Child("EmerCheckBox", 813);
  ALDTCheck := ASK specialBox Child("ALDTCheckBox", 814);
          70 LINES REMOVED
  ASK value6 TO DisplayValue(FLOAT(numRes));
  ASK value7 TO DisplayValue(ALDTcost);
  ASK value8 TO DisplayValue(Sparecost);
{The next lines do more checking on custom to see if checkboxes need be cheked or values disabled}
          107 LINES REMOVED
      ALDTVal := 0.;
      END IF;
      ALDTcost := value7.Value();
      Sparecost := value8.Value();
{Check to see if resources will be used and test for valid input}
      IF resCheck.Checked
          35 LINES REMOVED
          GET NODE TYPE INFORMATION
PROCEDURE GetNodeType(INOUT typeNode          : INTEGER;
              IN numLinksOut, numLinksIn : INTEGER;
              INOUT cancelledFlag      : BOOLEAN;
              INOUT ngcost,nycost,nrcost : REAL);
          9 LINES REMOVED
    radioBox := ASK nodeBox Child("NodeTypeRadBox",801);
    grncostBox := ASK nodeBox Child("GreenCost",814);
    ylwcostBox := ASK nodeBox Child("YellowCost", 815);
    redcostBox := ASK nodeBox Child("RedCost",816);
    {get a handle to the radio buttons}
    CASE typeNode
          20 LINES REMOVED
    ASK radioBox TO DisplaySelectedButton(ASK radioBox Child(buttonName, buttonNum));
    ASK grncostBox TO SetValue(ngcost);
    ASK ylwcostBox TO SetValue(nycost);
    ASK redcostBox TO SetValue(nrcost);
{display the selected button}
    ASK nodeBox TO GetNodeInput(typeNode, numLinksOut, numLinksIn, cancelledFlag, isStartNode,
                isEndNode, ngcost,nycost,nrcost);
    ngcost := grncostBox.Value;
    nycost := ylwcostBox.Value;
    nrcost := redcostBox.Value;
{get changes the user may have made--see nodeBoxObj in IDialogs}
    DISPOSE(nodeBox);
  END IF;
END PROCEDURE; {GetNodeType}
          PROCEDURE CLEAR OBJECT REMOVED
          EDIT DETAILS INFORMATION
PROCEDURE EditDetails;
VAR
  i,
  oldType,              {type of node before returning from DB}
  typeOfNode    : INTEGER;  {type of node: 1 = start, 2 = connect, 3 = stop}
  cancelled     : BOOLEAN;  {tells if this routine was cancelled}
  goodNodeValue : BOOLEAN;
  nodeTag       : STRING;   {# that appears below node (may not be same as node.Id)}
  goodPaths     : REAL;
  label         : LabelObj;
  value         : ValueBoxObj;
  button        : ButtonObj;
  nextNode      : RBDNodeObj;
  gcost, ycost, rcost     : REAL;
BEGIN
          30 LINES REMOVED
      oldType    := node.typeNode;
      gcost := node.ngcost;
      ycost := node.nycost;
```

```
      rcost := node.nrcost;
{For a node behavior is different when editing depending on whether the RBD is frozen or not}
      IF (NOT frozen)
{If not frozen you can change the type of node, so call the edit node dialog box}
        GetNodeType(typeOfNode, node.connectOutOfNum, node.connectIntoNum, cancelled,
              gcost,ycost,rcost);
        IF NOT cancelled
{If I didn't cancel}
          ASK node TO SetType(typeOfNode);
          ASK node TO SetNodeCost(gcost,ycost,rcost);
          IF typeOfNode = 1              {Start node}
            117 LINES REMOVED
            COPY OBJECT INFORMATION
            22 LINES REMOVED
{lets everyone know a block is copied}
  NEW(realArray, 1..6);
  realArray[1] := block.rradjfactor;
  realArray[2] := block.rrlimfactor;
  realArray[3] := block.radjfactor;
  realArray[4] := block.rlimfactor;
  realArray[5] := block.rrcost;
  realArray[6] := block.rcost;
  NEW(realArray2, 1..7);
  realArray2[1] := block.probAdj;
  realArray2[2] := block.uCostFactor;
  realArray2[3] := block.dCostFactor;
  realArray2[4] := block.fCostFactor;
  realArray2[5] := block.iCostFactor;
  realArray2[6] := block.sCostFactor;
  realArray2[7] := block.lFactor;
  ASK copyBlock TO SetBlockData(block.blockName, block.failDistro, block.numFailParams,
                    9, block.repairDistro, block.numRepairParams, 9,
                    block.arepairDistro, block.numARepairParams, 9,
                    block.failVals, block.repairVals, block.arepairVals, realArray,
                    realArray2, block.useAdj);
  DISPOSE(realArray);
  DISPOSE(realArray2);
{sets the copyBlock's parameters equal to those of the currently highlighted block, except for}
{random number streams--this is done on purpose to prevent user from having two blocks with the}
{same distribution and random stream, in which case they would always draw exactly the same}
{random numbers}
  NEW(realArray, 1..7);
  realArray[1]  := FLOAT(block.numSpares);
  realArray[2]  := block.spareRate;
  realArray[3]  := block.spareDelay;
  realArray[4]  := block.ALDTDelay;
  realArray[5]  := block.numPerVal;
  realArray[6]  := block.ALDTCostVal;
  realArray[7]  := block.SpareCostVal;
{Temporary array to use in sending the information to a set method/procedure.}
  NEW(boolsArray, 1..3);
  boolsArray[1] := block.systemDepend;
  boolsArray[2] := block.infiniteSpares;
  boolsArray[3] := block.useResource;
{Set the copyblock's special repair as well}
  ASK copyBlock TO SetSpecialData(realArray, boolsArray, block.poolName,
                    block.resourceName, block.numResources,
                    block.streamDepend, block.sparingType);
  DISPOSE(realArray);
  DISPOSE(boolsArray);
  ASK copyBlock TO SetConnectToNode(FALSE);
          6 LINES REMOVED
          PASTE OBJECT INFORMATION
          16 LINES REMOVED
    ASK root TO AddGraphic(block);
    NEW(realArray, 1..6);
```

```
    realArray[1] := copyBlock.rradjfactor;
    realArray[2] := copyBlock.rrlimfactor;
    realArray[3] := copyBlock.radjfactor;
    realArray[4] := copyBlock.rlimfactor;
    realArray[5] := copyBlock.rrcost;
    realArray[6] := copyBlock.rcost;
    NEW(realArray2, 1..7);
    realArray2[1] := copyBlock.probAdj;
    realArray2[2] := copyBlock.uCostFactor;
    realArray2[3] := copyBlock.dCostFactor;
    realArray2[4] := copyBlock.fCostFactor;
    realArray2[5] := copyBlock.iCostFactor;
    realArray2[6] := copyBlock.sCostFactor;
    realArray2[7] := copyBlock.lFactor;
    ASK block TO SetBlockData(copyBlock.blockName, copyBlock.failDistro,
                    copyBlock.numFailParams, copyBlock.failStream,
                    copyBlock.repairDistro, copyBlock.numRepairParams,
                    copyBlock.repairStream, copyBlock.arepairDistro,
                    copyBlock.numARepairParams, copyBlock.arepairStream,
                    copyBlock.failVals, copyBlock.repairVals, copyBlock.arepairVals,
                    realArray, realArray2, copyBlock.useAdj);
{set block's data equal to those of the copied block}
    DISPOSE(realArray);
    DISPOSE(realArray2);
    NEW(realArray, 1..7);
    realArray[1] := FLOAT(copyBlock.numSpares);
    realArray[2] := copyBlock.spareRate;
    realArray[3] := copyBlock.spareDelay;
    realArray[4] := copyBlock.ALDTDelay;
    realArray[5] := copyBlock.numPerVal;
    realArray[6] := copyBlock.ALDTCostVal;
    realArray[7] := copyBlock.SpareCostVal;
    NEW(boolsArray, 1..3);
    boolsArray[1] := copyBlock.systemDepend;
    boolsArray[2] := copyBlock.infiniteSpares;
    boolsArray[3] := copyBlock.useResource;
    ASK block TO SetSpecialData(realArray, boolsArray, copyBlock.poolName,
                    copyBlock.resourceName, copyBlock.numResources,
                    copyBlock.streamDepend, copyBlock.sparingType);
    DISPOSE(realArray);
    DISPOSE(boolsArray);
    ASK block TO SetConnectToNode(FALSE);
          17 LINES REMOVED
          SEND TO ENGINGE INFORMATION
          47 LINES REMOVED
  check11, check12,
  check13, check23, check24   : CheckBoxObj;
{generic check box objects}
          276 LINES REMOVED
    check8 := ASK simBox Child("RepairTimesCheckBox", 897);
    check23 := ASK simBox Child("MRRTFCheckBox", 950);
    check24 := ASK simBox Child("MARTFCheckBox", 951);
    check9 := ASK simBox Child("EventsCheckBox", 898);
          98 LINES REMOVED
  {set cursor to hourglass}
      GetSimOptions(check1.Checked, check2.Checked, check3.Checked, check4.Checked,
            check5.Checked, check6.Checked, check7.Checked, check8.Checked,
            check9.Checked, check10.Checked, check11.Checked, check12.Checked,
            check13.Checked, c22, check23.Checked, check24.Checked);
{this routine is in Iengine, all I am really doing is passing the state of all the sim options}
          104 LINES REMOVED
          SAVE DEFAULTS INFORMATION
          34 LINES REMOVED
  ASK defaultStream TO WriteString(" ");
  ASK defaultStream TO WriteInt(dRepStreamNum, 6);
  ASK defaultStream TO WriteString(" ");
```

```
ASK defaultStream TO WriteInt(dARepDistType, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteInt(dARepDistParms, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteInt(dARepStreamNum, 6);
ASK defaultStream TO WriteString(" ");
IF dSysDepen = FALSE
        16 LINES REMOVED
  ASK defaultStream TO WriteReal(dRepParmVals[i], 10, 6);
  ASK defaultStream TO WriteString(" ");
END FOR;
FOR i := 1 TO dARepDistParms
  ASK defaultStream TO WriteReal(dARepParmVals[i], 10, 6);
  ASK defaultStream TO WriteString(" ");
END FOR;
ASK defaultStream TO WriteLn;
IF dOperatingBlock
  ASK defaultStream TO WriteInt(1, 6);
ELSE
  ASK defaultStream TO WriteInt(0, 6);
END IF;
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(dProbSuccess, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(dUCost, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(dDCost, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(dFCost, 10, 6);
ASK defaultStream TO WriteReal(dICost, 10, 6);
ASK defaultStream TO WriteReal(dSCost, 10, 6);
ASK defaultStream TO WriteReal(dLevel, 10, 6);
ASK defaultStream TO WriteString(" ");
IF duseAdj
  ASK defaultStream TO WriteInt(1, 6);
ELSE
  ASK defaultStream TO WriteInt(0, 6);
END IF;
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(dprobAdj, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(drradjfactor, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(drrlimfactor, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(dradjfactor, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(drlimfactor, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(drrcost, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(drcost, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(dALDTCost, 10, 6);
ASK defaultStream TO WriteString(" ");
ASK defaultStream TO WriteReal(dSpareCost, 10, 6);
ASK defaultStream TO WriteString(" ");
{Additions features added with their appropriate default values, to include costing,
reliability growth/degradation and repair versus remove and replace maintenance strategy.}
IF dSpareType = SparePool
        73 LINES REMOVED
        OPEN DEFAULTS INFORMATION
        54 LINES REMOVED
ASK defaultStream TO ReadInt(dRepDistParms);
ASK defaultStream TO ReadInt(dRepStreamNum);
ASK defaultStream TO ReadInt(dARepDistType);
```

```
  ASK defaultStream TO ReadInt(dARepDistParms);
  ASK defaultStream TO ReadInt(dARepStreamNum);
  ASK defaultStream TO ReadInt(dep);
          18 LINES REMOVED
  ASK defaultStream TO ReadReal(dRepParmVals[i]);
  END FOR;
  NEW(dARepParmVals, 1..dARepDistParms);
  FOR i := 1 TO dARepDistParms
    ASK defaultStream TO ReadReal(dARepParmVals[i]);
  END FOR;
  ASK defaultStream TO ReadLine(sparing); {sparing here is dummy variable, checking for EOF}
{If End Of File is encountered, then this is a 2.0 file, set all 3.0 variables to defaults}
  IF defaultStream.eof
    raptor30        := FALSE;
    totalPools      := 0;
    dOperatingBlock  := TRUE;
    dProbSuccess     := 0.8;
    dUCost          := 0.0;
    dDCost          := 0.0;
    dFCost          := 0.0;
    dICost          := 0.0;
    dSCost          := 0.0;
    dLevel          := 1.0;
    duseAdj          := FALSE;
    dprobAdj         := 0.0;
    drradjfactor      := 1.0;
    drrlimfactor      := 2.0;
    dradjfactor := 1.0;
    drlimfactor := 2.0;
    drrcost := 0.0;
    drcost := 0.0;
    dALDTCost := 0.0;
    dSpareCost := 0.0;
    IF dInfinSpares
      dSpareType     := Infinite;
    ELSE
      dSpareType     := Custom;
    END IF;
    dPoolName        := "unnamed";
    dResName         := "unnamed";
    dUseResources     := FALSE;
    dNumRes          := 1;
  ELSE
    raptor30 := TRUE;
    ASK defaultStream TO ReadInt(operating);
    IF operating = 1
      dOperatingBlock := TRUE;
    ELSE
      dOperatingBlock := FALSE;
    END IF;
    ASK defaultStream TO ReadReal(dProbSuccess);
    ASK defaultStream TO ReadReal(dUCost);
    ASK defaultStream TO ReadReal(dDCost);
    ASK defaultStream TO ReadReal(dFCost);
    ASK defaultStream TO ReadReal(dICost);
    ASK defaultStream TO ReadReal(dSCost);
    ASK defaultStream TO ReadReal(dLevel);
    ASK defaultStream TO ReadInt(repbyadj);
    IF repbyadj = 1
      duseAdj := TRUE;
    ELSE
      duseAdj := FALSE;
    END IF;
    ASK defaultStream TO ReadReal(dprobAdj);
    ASK defaultStream TO ReadReal(drradjfactor);
    ASK defaultStream TO ReadReal(drrlimfactor);
```

```
  ASK defaultStream TO ReadReal(dradjfactor);
  ASK defaultStream TO ReadReal(drlimfactor);
  ASK defaultStream TO ReadReal(drrcost);
  ASK defaultStream TO ReadReal(drcost);
  ASK defaultStream TO ReadReal(dALDTCost);
  ASK defaultStream TO ReadReal(dSpareCost);
  ASK defaultStream TO ReadString(sparing);
  {Reading in the additional feature parameters.}
  IF sparing = "SparePool"
        62 LINES REMOVED
        PROCEDURE GET WINDOW PARAMETERS REMOVED
        PROCEDURE PRINT RBD REMOVED
        PRINT RESULTS INFORMATION
        68 LINES REMOVED
 printString := printString + tempPrint;
ASK printStream TO WriteString(printString);
ASK printStream TO WriteLn;
printString := FinalArray[1, 6] + "                ";
printString := SUBSTR(1, 20, printString);
tempPrint := SUBSTR(1, 15, (FinalArray[2, 6] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[3, 6] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[4, 6] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 12, (FinalArray[5, 6] + "        "));
printString := printString + tempPrint;
ASK printStream TO WriteString(printString);
ASK printStream TO WriteLn;
printString := FinalArray[1, 7] + "                ";
printString := SUBSTR(1, 20, printString);
tempPrint := SUBSTR(1, 15, (FinalArray[2, 7] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[3, 7] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[4, 7] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 12, (FinalArray[5, 7] + "        "));
printString := printString + tempPrint;
ASK printStream TO WriteString(printString);
ASK printStream TO WriteLn;
printString := FinalArray[1, 8] + "                ";
printString := SUBSTR(1, 20, printString);
tempPrint := SUBSTR(1, 15, (FinalArray[2, 8] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[3, 8] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[4, 8] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 12, (FinalArray[5, 8] + "        "));
printString := printString + tempPrint;
ASK printStream TO WriteString(printString);
ASK printStream TO WriteLn;
tempPrint := SUBSTR(1, 15, (FinalArray[2, 9] + "          ")) + " ";
printString := "% Green Time      " + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[3, 9] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[4, 9] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 12, (FinalArray[5, 9] + "        "));
printString := printString + tempPrint;
ASK printStream TO WriteString(printString);
ASK printStream TO WriteLn;
tempPrint := SUBSTR(1, 15, (FinalArray[2, 10] + "           ")) + " ";
printString := "% Yellow Time     " + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[3, 10] + "           ")) + " ";
```

```
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[4, 10] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 12, (FinalArray[5, 10] + "        "));
printString := printString + tempPrint;
ASK printStream TO WriteString(printString);
ASK printStream TO WriteLn;
tempPrint := SUBSTR(1, 15, (FinalArray[2, 11] + "          ")) + " ";
printString := "% Red Time        " + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[3, 11] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[4, 11] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 12, (FinalArray[5, 11] + "        "));
printString := printString + tempPrint;
ASK printStream TO WriteString(printString);
ASK printStream TO WriteLn;
printString := FinalArray[1, 12] + "                ";
printString := SUBSTR(1, 20, printString);
tempPrint := SUBSTR(1, 15, (FinalArray[2, 12] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[3, 12] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 15, (FinalArray[4, 12] + "          ")) + " ";
printString := printString + tempPrint;
tempPrint := SUBSTR(1, 12, (FinalArray[5, 12] + "        "));
printString := printString + tempPrint;
ASK printStream TO WriteString(printString);
ASK printStream TO WriteLn;
ASK printStream TO WriteLn;
ASK printStream TO WriteString(FinalArray[1, 1]);
ASK printStream TO WriteLn;
ASK printStream TO WriteLn;
IF StatsReset
  ASK printStream TO WriteString("Sparing Data shown is for the whole run (not reset)");
ELSE
  ASK printStream TO WriteString("Average sparing data over " + FinalArray[1, 2] + " run(s):");
END IF;
ASK printStream TO WriteLn;
ASK printStream TO WriteLn;
ASK printStream TO WriteString("COMPONENT      START      END      MIN      MAX      # DELAYS");
ASK printStream TO WriteLn;
ASK printStream TO
WriteString("_____");
ASK printStream TO WriteLn;
numInSpares := STRTOINT(FinalArray[1, 3]);
IF (numInSpares > 0)
  FOR p := 1 TO numInSpares
    tempPrint := SUBSTR(1, 16, (FinalArray[1, (p + 12)] + "          ")) + " ";
    printString := tempPrint;
    tempPrint := SUBSTR(1, 12, (FinalArray[2, (p + 12)] + "        ")) + " ";
    printString := printString + tempPrint;
    tempPrint := SUBSTR(1, 12, (FinalArray[3, (p + 12)] + "        ")) + " ";
    printString := printString + tempPrint;
    tempPrint := SUBSTR(1, 12, (FinalArray[4, (p + 12)] + "        ")) + " ";
    printString := printString + tempPrint;
    tempPrint := SUBSTR(1, 12, (FinalArray[5, (p + 12)] + "        ")) + " ";
    printString := printString + tempPrint;
    tempPrint := SUBSTR(1, 12, (FinalArray[6, (p + 12)] + "        ")) + " ";
    printString := printString + tempPrint;
    ASK printStream TO WriteString(printString);
    ASK printStream TO WriteLn;
  END FOR;
ELSE
  printString := "Sparing data not used";
  ASK printStream TO WriteString(printString);
```

```
  END IF;
  ASK printStream TO Close;
  DISPOSE(printStream);
END PROCEDURE; {PrintResults}
        DISPLAY RESULTS INFORMATION
        40 LINES REMOVED
  ASK resultTable TO SetText("MTBM", 1, 4);
  ASK resultTable TO SetText(FinalArray[1,6], 1, 5);
  ASK resultTable TO SetText(FinalArray[1,7], 1, 6);
  ASK resultTable TO SetText(FinalArray[1,8], 1, 7);
  ASK resultTable TO SetText("% Green Time", 1, 8);
  ASK resultTable TO SetText("% Yellow Time", 1, 9);
  ASK resultTable TO SetText("% Red Time", 1, 10);
  ASK resultTable TO SetText(FinalArray[1,12], 1, 11);
  ASK resultTable TO SetText(FinalArray[2,2], 2, 1);
  ASK resultTable TO SetText(FinalArray[3,2], 3, 1);
  ASK resultTable TO SetText(FinalArray[4,2], 4, 1);
  ASK resultTable TO SetText(FinalArray[5,2], 5, 1);
  ASK resultTable TO SetText(FinalArray[2,3], 2, 2);
  ASK resultTable TO SetText(FinalArray[3,3], 3, 2);
  ASK resultTable TO SetText(FinalArray[4,3], 4, 2);
  ASK resultTable TO SetText(FinalArray[5,3], 5, 2);
  ASK resultTable TO SetText(FinalArray[2,4], 2, 3);
  ASK resultTable TO SetText(FinalArray[3,4], 3, 3);
  ASK resultTable TO SetText(FinalArray[4,4], 4, 3);
  ASK resultTable TO SetText(FinalArray[5,4], 5, 3);
  ASK resultTable TO SetText(FinalArray[2,5], 2, 4);
  ASK resultTable TO SetText(FinalArray[3,5], 3, 4);
  ASK resultTable TO SetText(FinalArray[4,5], 4, 4);
  ASK resultTable TO SetText(FinalArray[5,5], 5, 4);
  ASK resultTable TO SetText(FinalArray[2,6], 2, 5);
  ASK resultTable TO SetText(FinalArray[3,6], 3, 5);
  ASK resultTable TO SetText(FinalArray[4,6], 4, 5);
  ASK resultTable TO SetText(FinalArray[5,6], 5, 5);
  ASK resultTable TO SetText(FinalArray[2,7], 2, 6);
  ASK resultTable TO SetText(FinalArray[3,7], 3, 6);
  ASK resultTable TO SetText(FinalArray[4,7], 4, 6);
  ASK resultTable TO SetText(FinalArray[5,7], 5, 6);
  ASK resultTable TO SetText(FinalArray[2,8], 2, 7);
  ASK resultTable TO SetText(FinalArray[3,8], 3, 7);
  ASK resultTable TO SetText(FinalArray[4,8], 4, 7);
  ASK resultTable TO SetText(FinalArray[5,8], 5, 7);
  ASK resultTable TO SetText(FinalArray[2,9], 2, 8);
  ASK resultTable TO SetText(FinalArray[3,9], 3, 8);
  ASK resultTable TO SetText(FinalArray[4,9], 4, 8);
  ASK resultTable TO SetText(FinalArray[5,9], 5, 8);
  ASK resultTable TO SetText(FinalArray[2,10], 2, 9);
  ASK resultTable TO SetText(FinalArray[3,10], 3, 9);
  ASK resultTable TO SetText(FinalArray[4,10], 4, 9);
  ASK resultTable TO SetText(FinalArray[5,10], 5, 9);
  ASK resultTable TO SetText(FinalArray[2,11], 2, 10);
  ASK resultTable TO SetText(FinalArray[3,11], 3, 10);
  ASK resultTable TO SetText(FinalArray[4,11], 4, 10);
  ASK resultTable TO SetText(FinalArray[5,11], 5, 10);
  ASK resultTable TO SetText(FinalArray[2,12], 2, 11);
  ASK resultTable TO SetText(FinalArray[3,12], 3, 11);
  ASK resultTable TO SetText(FinalArray[4,12], 4, 11);
  ASK resultTable TO SetText(FinalArray[5,12], 5, 11);
  spareTable := ASK dialogBox Child("SparesTable", 715);
  ASK spareTable TO SetHeadings(FALSE, TRUE);
  ASK spareTable TO SetText("COMPONENT", 1, 0);
  ASK spareTable TO SetText("START", 2, 0);
  ASK spareTable TO SetText("END", 3, 0);
  ASK spareTable TO SetText("MIN", 4, 0);
  ASK spareTable TO SetText("MAX", 5, 0);
  ASK spareTable TO SetText("# DELAYS", 6, 0);
```

```
   numInSpares := STRTOINT(FinalArray[1,3]);
   IF (numInSpares > 0)
{if I have some spares data to display}
     ASK spareTable TO SetSize(6 , numInSpares);
{sizes the spares table for the number of items I will be displaying}
     FOR p := 1 TO numInSpares
       ASK spareTable TO SetText(FinalArray[1, (p + 12)], 1, p);
{starts to fill the table with data}
       ASK spareTable TO SetText(FinalArray[2, (p + 12)], 2, p);
       ASK spareTable TO SetText(FinalArray[3, (p + 12)], 3, p);
       ASK spareTable TO SetText(FinalArray[4, (p + 12)], 4, p);
       ASK spareTable TO SetText(FinalArray[5, (p + 12)], 5, p);
       ASK spareTable TO SetText(FinalArray[6, (p + 12)], 6, p);
     END FOR;
   ELSE
          22 LINES REMOVED
          PROCEDURE SELECT BLOCK REMOVED
          PROCEDURE SELECT NODE REMOVED
          PROCEDURE SET ZOOM REMOVED
          PROCEDURE INIT LIST REMOVED
          PROCEDURE SHOW ME NODE REMOVED
          PROCEDURE CHECK OPEN FILE STATUS REMOVED
          MAIN MENU OBJ INFORMATION
OBJECT mainMenuObj;
          BE SELECTED INFORMATION
          358 LINES REMOVED
        DISPOSE(dRepParmVals);
        DISPOSE(dARepParmVals);
        DISPOSE(root);
          102 LINES REMOVED
{dialog box}
      GetBlockName(dBlockName, cancelled, remains, dSysDepen, dOperatingBlock,
            dFailStreamNum, dRepStreamNum, dARepStreamNum, dStrmDepen,
            dProbSuccess, dUCost, dSCost, dDCost, dFCost, dICost, dLevel);
{calls the same procedures as if adding a block, but stores user's values in default variables}
{instead of a block object}
{If I didn't cancel and the default block is not an event block, then get the rest}
      IF NOT cancelled AND dOperatingBlock
        NEW(realArray, 1..6);
        realArray[1] := drradjfactor;
        realArray[2] := drrlimfactor;
        realArray[3] := dradjfactor;
        realArray[4] := drlimfactor;
        realArray[5] := drrcost;
        realArray[6] := drcost;
        InitGetDistro(dBlockName, dFailDistType, dFailDistParms, dRepDistType,
              dRepDistParms, dARepDistType, dARepDistParms,
              dFailParmVals, dRepParmVals, dARepParmVals, realArray,
              dprobAdj, duseAdj);
        drradjfactor := realArray[1];
        drrlimfactor := realArray[2];
        dradjfactor  := realArray[3];
        drlimfactor  := realArray[4];
        drrcost      := realArray[5];
        drcost       := realArray[6];
        DISPOSE(realArray);
        distLabel := ("Default Maintenance Information");
        GetSpecial(distLabel, dNumSpares, dSparesReplen, dSparesDelay, dALDTVal,
              dALDTCost, dSpareCost,
              dnumPerVal, dInfinSpares, dUseResources, dPoolName,
              dResName, dNumRes, dSpareType);
        somethingChanged := TRUE;
      END IF;
      Enable2Thru6;
          329 LINES REMOVED
          METHOD DISABLE REMOVED
```

```
            METHOD DISABLE 1 THRU 8 REMOVED
            METHOD ENABLE REMOVED
            METHOD ENABLE 2 THRU 5 REMOVED
            METHOD ENABLE 2 THRU 6 REMOVED
            METHOD ADD FILE NAME REMOVED
            METHOD SET FILE NAMES REMOVED
END OBJECT; {mainMenuObj}
            MAIN WINDOW OBJ INFORMATION
OBJECT mainWindowObj;
            METHOD BE SCROLLED REMOVED
            METHOD BE CLOSED REMOVED
            METHOD OBJINIT REMOVED
            MOUSE CLICK INFORMATION
            20 LINES REMOVED
    repStreamNum,
    arepDistType, arepDistParms, arepStreamNum,
{repair stream number}
            20 LINES REMOVED
    probSuccess,
    upCost,
    downCost,
    failCost,
    initialCost,
    standCost,
    level,
    pAdjust,
    rradjfact,
    rrlimfact,
    radjfact,
    rlimfact,
    gcost,
    ycost,
    rcost,
    ALDTcost,
    Sparecost,
    rrcostval,
    rcostval   : REAL;
{number of replenishment spares}
    cancelled,
            8 LINES REMOVED
    operates,
    uAdjust   : BOOLEAN;
{tells if a block has infinite spares}
    blockImage,
    selectedObj   : ImageObj;
{the object which is currently selected}
    failParmVals,
    repParmVals,
    arepParmVals,
    eventParam    : valueArray;
            7 LINES REMOVED
    numRes,
    strmDepen     : INTEGER;
    spareType     : SparingType;
  BEGIN
            180 LINES REMOVED
            blockName := dBlockName;
{set preliminary block info equal to the defaults}
            sysDepen := dSysDepen;
            strmDepen := dStrmDepen;
            operates := dOperatingBlock;
            failStreamNum := dFailStreamNum;
            repStreamNum := dRepStreamNum;
            arepStreamNum := dARepStreamNum;
            probSuccess := dProbSuccess;
            upCost := dUCost;
```

```
            downCost := dDCost;
            failCost := dFCost;
            initialCost := dICost;
            {Additional features added into the program.}
            GetBlockName(blockName, cancelled, remains, sysDepen, operates,
                  failStreamNum, repStreamNum, arepStreamNum, strmDepen,
                  probSuccess, upCost, standCost, downCost,
                  failCost, initialCost, level);
{see if user wants to change any of the defaults}
            IF cancelled
            8 LINES REMOVED
              eventParam[1] := probSuccess;
              NEW(realArray, 1..6);
              realArray[1] := 1.0;
              realArray[2] := 2.0;
              realArray[3] := 1.0;
              realArray[4] := 2.0;
              realArray[5] := 0.0;
              realArray[6] := 0.0;
              NEW(realArray2, 1..7);
              realArray2[1] := pAdjust;
              realArray2[2] := upCost;
              realArray2[3] := downCost;
              realArray2[4] := failCost;
              realArray2[5] := initialCost;
              realArray2[6] := standCost;
              realArray2[7] := level;
              ASK block TO SetBlockData(blockName, 17, 1, failStreamNum, 17, 1, failStreamNum,
                        17, 1, failStreamNum, eventParam, eventParam, eventParam,
                        realArray, realArray2, FALSE);
              DISPOSE(realArray);
              DISPOSE(realArray2);
              NEW(realArray, 1..7);
              NEW(boolsArray, 1..3);
              boolsArray[1] := sysDepen;
              ASK block TO SetSpecialData(realArray, boolsArray, "unnamed", "unnamed", 0,0, None);
              DISPOSE(realArray);
            8 LINES REMOVED
        { set the rest of the block's info equal to the defaults}
              NEW(realArray, 1..7);
              realArray[1] := dNumSpares;
              realArray[2] := dSparesReplen;
              realArray[3] := dSparesDelay;
              realArray[4] := dALDTVal;
              realArray[5] := dnumPerVal;
              realArray[6] := dALDTCost;
              realArray[7] := dSpareCost;
              NEW(boolsArray, 1..3);
              boolsArray[1] := sysDepen;
              boolsArray[2] := dInfinSpares;
              boolsArray[3] := dUseResources;
              ASK block TO SetSpecialData(realArray, boolsArray, dPoolName,
                        dResName, dNumRes, strmDepen,dSpareType);
              DISPOSE(realArray);
              DISPOSE(boolsArray);
              NEW(realArray, 1..6);
              realArray[1] := drradjfactor;
              realArray[2] := drrlimfactor;
              realArray[3] := dradjfactor;
              realArray[4] := drlimfactor;
              realArray[5] := drrcost;
              realArray[6] := drcost;
              NEW(realArray2, 1..7);
              realArray2[1] := dprobAdj;
              realArray2[2] := upCost;
              realArray2[3] := downCost;
```

```
            realArray2[4] := failCost;
            realArray2[5] := initialCost;
            realArray2[6] := standCost;
            realArray2[7] := level;
            {Temporary arrays to provide input/output to the specific methods/procedures.}
            ASK block TO SetBlockData(blockName, dFailDistType,
                        dFailDistParms, failStreamNum, dRepDistType,
                        dRepDistParms, repStreamNum, dARepDistType,
                        dARepDistParms, arepStreamNum, dFailParmVals,
                        dRepParmVals, dARepParmVals, realArray,
                        realArray2,duseAdj);
            DISPOSE(realArray);
            DISPOSE(realArray2);
            somethingChanged := TRUE;
          14 LINES REMOVED
            repParmVals[i] := dRepParmVals[i];
            END FOR;
            arepDistType := dARepDistType;
            arepDistParms := dARepDistParms;
            NEW(arepParmVals, 1..arepDistParms);
            FOR i := 1 TO arepDistParms
              arepParmVals[i] := dARepParmVals[i];
            END FOR;
            pAdjust  := dprobAdj;
            uAdjust  := duseAdj;
            rradjfact := drradjfactor;
            rrlimfact := drrlimfactor;
            radjfact  := dradjfactor;
            rlimfact  := drlimfactor;
            rrcostval    := drrcost;
            rcostval     := drcost;
            NEW(realArray, 1..6);
            realArray[1]  := rradjfact;
            realArray[2]  := rrlimfact;
            realArray[3]   := radjfact;
            realArray[4]   := rlimfact;
            realArray[5]   := rrcostval;
            realArray[6]   := rcostval;
{Call InitGetDistro procedure in this module}
            InitGetDistro(blockName, failDistType, failDistParms, repDistType,
                    repDistParms, arepDistType, arepDistParms,
                    failParmVals, repParmVals, arepParmVals, realArray,
                    pAdjust, uAdjust);
            numSpares := dNumSpares;
            rradjfact := realArray[1];
            rrlimfact := realArray[2];
            radjfact := realArray[3];
            rlimfact := realArray[4];
            rrcostval := realArray[5];
            rcostval  := realArray[6];
            DISPOSE(realArray);
{then get the special repair info}
            sparesReplen := dSparesReplen;
            sparesDelay := dSparesDelay;
            ALDTVal := dALDTVal;
            numPerVal := dnumPerVal;
            infinSpares := dInfinSpares;
            useResource := dUseResources;
            poolName := dPoolName;
            resName := dResName;
            numRes := dNumRes;
            spareType := dSpareType;
            ALDTcost := dALDTCost;
            Sparecost := dSpareCost;
            distLabel := (blockName + "'s Maintenance Information");
{Call dialog box to get special repair info}
```

```
            GetSpecial(distLabel, numSpares, sparesReplen, sparesDelay, ALDTVal,
                  ALDTcost, Sparecost, numPerVal,
                  infinSpares, useResource, poolName, resName, numRes, spareType);
            NEW(realArray, 1..7);
            realArray[1] := numSpares;
            realArray[2] := sparesReplen;
            realArray[3] := sparesDelay;
            realArray[4] := ALDTVal;
            realArray[5] := numPerVal;
            realArray[6] := ALDTcost;
            realArray[7] := Sparecost;
            NEW(boolsArray, 1..3);
            boolsArray[1] := sysDepen;
            boolsArray[2] := infinSpares;
            boolsArray[3] := useResource;
{these next lines embed the info within the block object iteself}
            ASK block TO SetSpecialData(realArray, boolsArray, poolName,
                        resName, numRes, strmDepen, spareType);
            DISPOSE(realArray);
            DISPOSE(boolsArray);
            NEW(realArray, 1..6);
            realArray[1] := rradjfact;
            realArray[2] := rrlimfact;
            realArray[3] := radjfact;
            realArray[4] := rlimfact;
            realArray[5] := rrcostval;
            realArray[6] := rcostval;
            NEW(realArray2, 1..7);
            realArray2[1] := pAdjust;
            realArray2[2] := upCost;
            realArray2[3] := downCost;
            realArray2[4] := failCost;
            realArray2[5] := initialCost;
            realArray2[6] := standCost;
            realArray2[7] := level;
            ASK block TO SetBlockData(blockName, failDistType,
                        failDistParms, failStreamNum, repDistType,
                        repDistParms, repStreamNum, arepDistType,
                        arepDistParms, arepStreamNum, failParmVals,
                        repParmVals, arepParmVals, realArray, realArray2,
                        uAdjust);
            DISPOSE(realArray);
            DISPOSE(realArray2);
            DISPOSE(failParmVals);
            DISPOSE(repParmVals);
            DISPOSE(arepParmVals);
            somethingChanged := TRUE;
          31 LINES REMOVED
          typeOfNode := 2;
          gcost := 0.0;
          ycost := 0.0;
          rcost := 0.0;
{set 'connect node' as the default}
          cancelled := FALSE;
          GetNodeType(typeOfNode, node.connectOutOfNum, node.connectIntoNum, cancelled,
                gcost, ycost, rcost);
{checks to see if user wants to change default node type}
          6 LINES REMOVED
            ASK node TO SetType(typeOfNode);
            ASK node TO SetNodeCost(gcost,ycost,rcost);
            IF typeOfNode = 1
          294 LINES REMOVED
END OBJECT; {mainWindowObj}
          ALL OUTPUT WINDOW OBJ REMOVED
END MODULE. {imod rbdwin}
```

CHAPTER 4

ADVANCING RM&A ANALYSIS

Introduction

Chapter 4 comprises three projects advancing the study of reliability, maintainability, and availability (RM&A) analysis. The enhanced version of RAPTOR (Rapid Availability Prototyping for Testing Operational Readiness) provides the simulation results for all three case studies. Each case study provides contribution to the field of reliability engineering. The sections of this chapter form the basis for three articles to be submitted for publication in the literature of reliability analysis.

The first case study deals with using the triangle distribution in place of the Weibull distribution when little or no data is available. The triangle distribution is estimated using zero as the minimum and the most likely and worst case estimates as the mode and maximum parameters. The estimates are derived from engineering judgement and a distribution to model a component's time to failure. The sensitivity of the output is tested by comparing the simulation results using the triangle distribution to the simulation results using the Weibull (the assumed, true distribution) distribution.

The second section presents a case study comparing sparing strategies for a communications network. The alternatives for sparing are derived from manipulating the time to replace emergency spares. Three alternatives are evaluated based on maintainability costs. The cost values are adjusted to take into consideration the time value of money, with the communication network existing over 15 years. This section provides insight into using simulation, coupled with spreadsheet analysis, as a decision-

making tool. The end result is a trade-off analysis with advantages and disadvantages listed for each alternative.

The final section of this Chapter deals with clarification of the time between failure and time to failure distributions. These distributions are often confused in the literature and misapplied in system analysis. The case study presents a detailed discussion on using the appropriate distribution and the errors resulting when the wrong distribution is implemented. The scenario focuses on system availability and predicting the sparing requirements to sustain a fleet of automobiles. The simulations demonstrate the affect of using a time between failure distribution, when in fact, the time to failure distribution should be used.

The three case studies in this chapter advance the understanding of RM&A and demonstrate the utility of reliability simulation. The scenarios also indicate the relative ease of using reliability simulation for the practicing engineer. Each case study lays the groundwork for publishing original contributions in the field of reliability engineering. The detailed analysis advances RM&A analysis or clarifies areas of confusion in the literature. In any case, the scenarios are presented to validate the potential use of reliability simulation.

Using the Triangle Distribution

INTRODUTION

System level reliability, maintainability and availability (RM&A) analysis supports product and process improvement through increased understanding of system performance. The Department of Defense and the United States Air Force rely heavily

on RM&A analysis during test and evaluation in the acquisition of new weapons and support equipment. Reduced budget and limited manpower requires alternative decision making tools when evaluating new system performance. The industrial community further supports RM&A analysis as a means to improve products and processes to remain competitive in a global economy.

Reliability simulation provides one alternative for performing RM&A analysis on complex systems. The ability of current simulation software to predict system behavior allows for efficient analysis using modeling to report system performance. The limitation to the effectiveness of the simulation is the quality of the input. Because of its versatility, the Weibull distribution is often used to characterize component times to failure. The scale and shape parameters must often be approximated with little or no component historical data. Poor approximations will affect the output of the simulation.

Durkee, et al (1997) examined the possibility of using the exponential distribution in place of the Weibull when little data was available for approximating the parameters of the distribution. They found the exponential distribution could be used during simulation and would give accurate availability predictions. The component's failure times were between 1000 and 5000 hours. The system availability was set high (96.6%) and thus reduced the sensitivity to differences in the input distributions. The triangle distribution would simplify things one step further. Engineers can easily describe the three parameters for the distribution. Law and Kelton (1991) refer to these as the "most optimistic and pessimistic estimates", along with a "subjective estimate of the most likely time to perform the task". In the case of failure times, the minimum value is set to zero,

the mode represents the most likely value, and the maximum represents the best case scenario. This makes defining the component failure distribution more intuitive to the user of the simulation software. The accuracy of the triangle distribution is affected by "subjective estimates of the absolute minimum and maximum possible values" (Law and Kelton (1991)). However, with little or no data available, this may be the best possible approach to provide an estimate of system performance.

SCOPE

The goal of this research is to obtain accurate predictions for complex system behavior. Specifically, the input distributions are manipulated when little or no data is available. The Weibull distribution will be replaced by the triangle distribution to determine the effect on system availability and mean time to first failure. The first step will be to simulate the true system performance, using Weibull failure distributions with perfect knowledge of the two parameters. Next, the component's failure distribution will be replaced with the triangle distribution with vertices at 0, the Weibull mode and an optimal high percentile obtained experimentally. A designed experiment will be used to perform a sensitivity analysis as the triangle distribution mode and maximum are shifted away from their best values. Our final analysis will determine the extent to which a triangle distribution may be used to approximate each component's Weibull failure distribution for a complex system.

BACKGROUND – RELIABILITY BLOCK DIAGRAMS

The structure of the system will be described using reliability block diagrams. The five, ten and fifteen component systems used during this analysis are shown below.

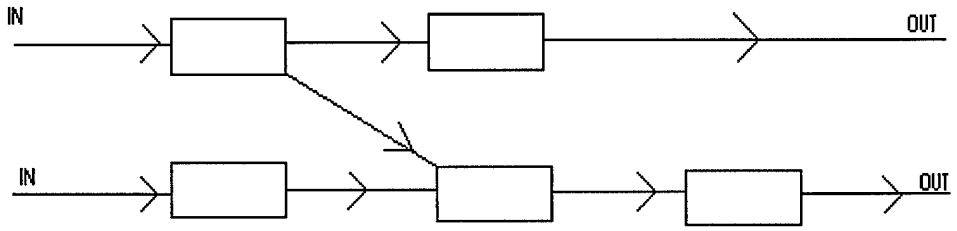


FIGURE 4-1. Reliability Block Diagram for Five-component System

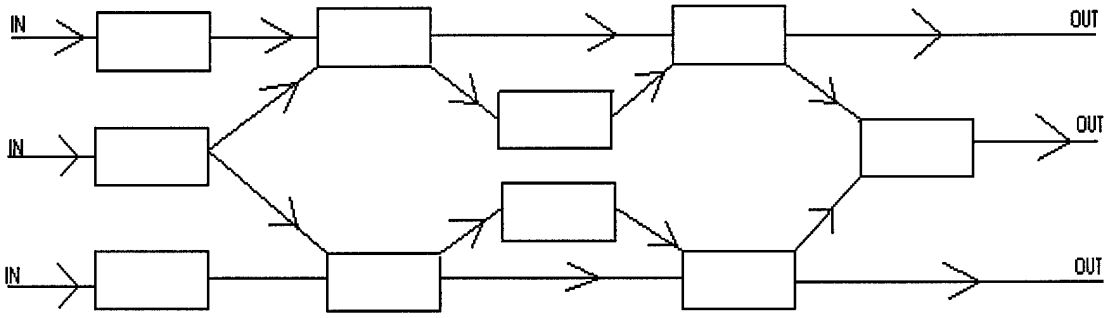


FIGURE 4-2. Reliability Block Diagram for Ten-component System

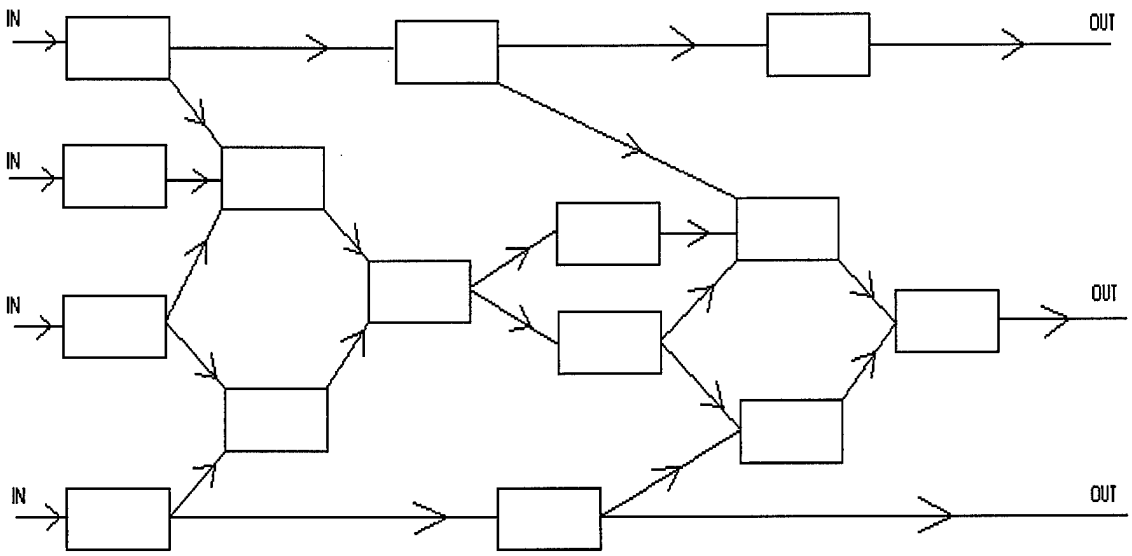


FIGURE 4-3. Reliability Block Diagram for Fifteen-component System

For each of the five components above the failure and repair distribution must be specified to accurately model the system. The simulation software will model the user-defined information to predict the availability or reliability of the system. Similarly, a ten and fifteen component system are developed and used during the analysis.

RELIABILITY SIMULATION

The simulation software used in this research was developed by the United States Air Force. The Rapid Availability Prototyping for Testing Operational Readiness (RAPTOR™) software (1995) was developed the Air Force Operational Test and Evaluation Center (AFOTEC) for use in new program evaluation. RAPTOR™ is built with a graphical user interface and allows the user to construct the RBD in a point-and-click manner. RAPTOR™ is available in the public domain at no charge. The component behavior distributions are input through prompted menus. Any one of fifteen

distributions can be defined, including the Weibull and the triangle distributions. The software was used to perform all the simulations in this analysis.

SIMULATION THE TRUE SYSTEM BEHAVIOR – SYSTEM ASSUMPTIONS

The system is composed of components connected in a complex manner to model system construction. Each component is defined with individual distributions for failure and repair. The failure distributions are Weibull with alpha (scale) and beta (shape) parameters ranging from 1.2 to 2.8 and 2200 to 3400 respectively. The reliability of the individual components ranges from 50% to 90%. The repair distributions are lognormal and the parameters are adjusted accordingly to maintain the desired reliability for each individual component. Failed components are repaired when evaluating system availability. However, no repairs are completed during the mean time to first failure analysis in the second part of the study.

All three systems (five, ten and fifteen components) are used during the evaluation. The components stop operating when the system is in a failed condition. They are considered in stand-by while the failed critical component is repaired. No additional hours accumulate on the components. There are an infinite number of spares for repairing components as they fail. The repair action returns the failed component to a "good as new" condition. The primary metric of interest is the overall availability of the system and mean time to first failure (when no repairs are completed).

Shown below is a table of five components and their failure and repair distribution parameters.

Component Number	Alpha for failure distribution	Beta for failure distribution	Mean for repair distribution	Std dev for repair distribution
Number 1	1.2	2200	517	51.7
Number 2	1.6	3400	3048	304.8
Number 3	2.0	1800	1063	106.3
Number 4	2.4	2600	256	25.6
Number 5	2.8	3000	1145	114.5

TABLE 4-1. Failure and Repair Distributions for Individual Components

The ten and fifteen component systems used two and three of each of the above components respectively.

RESULTS FOR AVAILABILITY AND MEAN TIME TO FIRST FAILURE

The RAPTOR™ simulation program can be used to model both availability and mean time to first failure. During the availability evaluation, all components have infinite spares and the system is simulation for 1,000,000 operating hours. The overall availability is calculated by the software and reported in the output report for each simulation. A second simulation used different random number streams to calculate an average availability, which will represent the "true" performance of the system.

During the mean time to first failure (MTTFF) evaluation, the simulation ran until the system failed. This was repeated 2,000 times to obtain an accurate system time to

failure estimate and was repeated with a different set of random number streams. Shown below is the table of results for the true five, ten and fifteen component system.

Size	Avail run 1	Avail run 2	Avail run 3	Avg avail	MTTFF run 1	MTTFF run 2	MTTFF run 3	Avg MTTFF
5	77.5	77.0	78.4	77.63	1872	1867	1911	1883.3
10	56.7	57.5	57.5	57.23	1701	1656	1684	1680.3
15	87.6	87.3	87.4	87.43	2170	2157	2194	2173.7

TABLE 4-2. Availability and MTTFF For True 5, 10, and 15 Component Systems

Availability is based on the percentage of uptime during the 1,000,000 operating hours, while the mean time to failure is based on the number of hours the system survives before experiencing the first failure bringing the system to a failed condition.

USING A TRIANGLE DISTRIBUTION – ADJUSTING THE MAXIMUM VALUE

The failure distribution for all components is changed to triangle distribution. The parameters for the triangle are calculated based on the true Weibull distribution. The minimum value is set equal to zero, the mode of the triangle is set equal to the mode of the Weibull distribution. The maximum value is adjusted according to the high-end percentiles of the Weibull distribution. Initially, the 85^{th} percentile is used and the simulations are completed to compare the output results.

The availability of the five component system serves as the performance metric of interest to determine the setting of the triangle distribution maximum value. As the maximum value changes from the 85th percentile to the 90th percentile, the simulated availability increases closer to the true (availability using Weibull failure distributions) system behavior. The four percentiles used as maximum values for the triangle distribution are 85, 90, 95 and 98th percentile. The simulation results for availability are 69.2%, 72.4%, 74.8% and 78.6% respectively. The true system availability is 77.63%, thus justifying use of the 98th percentile for anchoring the maximum value of the triangle distribution. Similar simulations were performed with the ten and fifteen block system and using mean time to failure as the metric of interest. In all but one case, the 98th percentile gave the closest performance to the true system.

SENSITIVITY ANALYSIS WHEN CHANGING THE DISTRIBUTION

The next step in the analysis is to conduct a sensitivity analysis by changing the mode and maximum of the triangle distribution to determine the effect on the output results. The minimum value is left anchored at zero. The mode and the maximum values are varied by +/- 5%, 10%, 20% and 30%. The goal of this analysis is to determine how sensitive the availability and MTTFF performance metrics are to changes in the failure distribution specification.

The following table represents simulation output from all three systems with respect to availability. The mode and the maximum are adjusted by all four percentages and the minimum value is left fixed at zero. The relative error is calculated based on the

true system performance using the Weibull failure distribution. Mode equal to high or low implies the mode shifted up or down by the appropriate percentage, similarly for the maximum value. All cases were replicated three times using different random numbers.

Size	Range	Mode, High Max, High	Mode, High Max, Low	Mode, Low Max, High	Mode, Low Max, Low
5	+/- 5%	2.37, 2.77, 3.27	0.97, -0.23, 0.57	2.37, 2.07, 1.67	0.77, -0.33, -0.03
5	+/- 10%	3.87, 3.37, 4.07	-0.43, -0.03, 1.07	2.17, 2.47, 2.17	-0.93, -1.03, -1.93
5	+/- 20%	5.57, 4.37, 4.57	-0.73, -2.03, -1.13	2.97, 3.57, 2.67	-4.43, -5.33, -4.03
5	+/- 30%	6.27, 7.17, 7.67	-3.83, -3.43, -2.83	3.97, 5.77, 3.77	-8.13, -7.93, -8.43
10	+/- 5%	4.07, 2.57, 2.87	-0.13, 1.17, 1.27	0.37, 2.27, 1.67	0.77, -0.23, 0.37
10	+/- 10%	3.67, 5.47, 5.67	-0.13, -0.53, -1.73	1.77, 3.87, 2.77	-0.73, -3.83, -2.43
10	+/- 20%	7.97, 7.67, 6.97	-3.13, -2.63, -2.23	4.87, 5.87, 4.57	-6.03, -6.23, -4.73
10	+/- 30%	8.57, 9.37, 8.47	-4.13, -2.83, -4.63	6.37, 5.17, 4.77	-9.33, -9.93, -10.13
15	+/- 5%	2.57, 1.87, 2.07	0.67, 0.47, 0.47	2.17, 1.47, 1.67	0.77, 0.37, 0.27
15	+/- 10%	2.67, 2.77, 3.37	0.17, 0.27, -0.73	2.17, 1.17, 1.77	-2.53, -0.93, -0.23
15	+/- 20%	3.97, 3.97, 4.47	-0.93, -1.43, -1.33	3.07, 2.57, 3.17	-4.13, -4.13, -4.13
15	+/- 30%	5.47, 5.47, 5.17	-2.43, -1.93, -1.83	3.87, 3.47, 3.37	-7.13, -6.93, -7.73

- Absolute error = Expected – Experimental
- Expected = Average (of 3 reps) simulated availability using Weibull failure distribution
- Experimental = Simulated availability using triangle distribution

TABLE 4-3. Absolute Errors for System Availability Using the Triangle Distribution

As expected, the greater deviation from the distribution parameters resulted in a greater deviation from the true availability. The deviation, however, is not excessive compared to the amount of deviation from the triangle parameters. Analyzing the results demonstrate the lack of sensitivity of the availability metric to the failure distribution specification. Incorrectly specifying the mode and maximum by 30% changes the availability percentage by less than 10%. This gives the user confidence in evaluating complex system behavior with marginally accurate failure distribution estimates. The original triangle distribution (with min equal to zero, mode equal to Weibull mode, and max equal to Weibull 98^{th} percentile) is also simulated and returned a relative error of –1.79, -5.93, and –2.32 percent for all three systems (five, ten and fifteen components).

Similar results can be seen in the following table using MTTFF as the response variable of interest. The change in mean time to failure is divided by the true value to calculate the relative error. Absolute error was used as the metric during the availability analysis since the parameter is already expressed as a percentage. All three systems are analyzed along with the same percentage deviations in the input distributions.

Size	Range	Mode, High Max, High	Mode, High Max, Low	Mode, Low Max, High	Mode, Low Max, Low
5	+/- 5%	8.74, 9.75, 7.73	1.84, 2.85, 0.94	5.24, 7.58, 4.23	-1.66, 0.62, -2.51
5	+/- 10%	13.89, 20.16, 12.83	0.14, 1.10, -0.76	6.88, 7.95, 5.77	-6.81, -5.91, -7.66
5	+/- 20%	24.25, 25.47, 23.13	-3.52, -2.62, -4.32	10.23, 11.29, 8.90	-17.17, -16.37, -17.91
5	+/- 30%	34.60, 35.93, 33.38	-7.50, -6.71, -8.35	13.58, 14.69, 12.04	-27.52, -26.83, -28.16
10	+/- 5%	10.39, 11.47, 11.76,	2.96, 3.91, 4.21	7.30, 8.37, 8.61	-0.14, 0.87, 1.11
10	+/- 10%	15.63, 16.76, 17.06	0.69, 1.65, 1.88	9.38, 10.51, 10.81	-5.38, -4.42, -4.19
10	+/- 20%	26.17, 27.42, 27.77	-3.89, -3.11, -2.82	13.55, 14.74, 15.04	-15.91, -15.08, -14.84
10	+/- 30%	36.70, 38.01, 38.37	-8.65, -7.99, -7.76	17.66, 19.02, 19.26	-26.38, -26.26, -25.49
15	+/- 5%	7.74, 8.20, 6.36	0.98, 1.35, -0.31	4.25, 4.75, 2.82	-2.51, -2.10, -3.80
15	+/- 10%	12.90, 13.36, 11.38	-0.67, -0.40, -1.92	5.86, 6.46, 4.39	-7.62, -7.25, -8.86
15	+/- 20%	23.16, 23.66, 21.50	-4.17, -3.99, -5.37	9.12, 9.86, 7.56	-17.88, -17.56, -18.98
15	+/- 30%	33.42, 33.97, 31.67	-8.04, -7.90, -9.19	12.44, 13.40, 10,83	-28.19, -27.86, -29.11

- Relative Error = ((Expected – Experimental) / Expected) * 100
- Expected = Average (of 3 runs) simulated MTTFF using Weibull distribution
- Experimental = Simulated MTTFF using triangle distribution

TABLE 4-4. Relative Errors for System MTTFF Using the Triangle Distribution

The results for the mean time to failure indicate a performance metric more sensitive than availability. The relative error for MTTFF when incorrectly specifying the triangle parameters increases to over 30%. The most dramatic cases are seen when the mode and maximum are both high or both low. As an example, in the fifteen component system having both parameters high led to a 34% absolute error. The simulated MTTFF is

approximately 2900 hours versus a true MTTFF of only 2160 hours. Similarly, having both parameters specified low, the MTTFF is 1565 hours versus the same true performance of 2160 hours resulting in 28% absolute error.

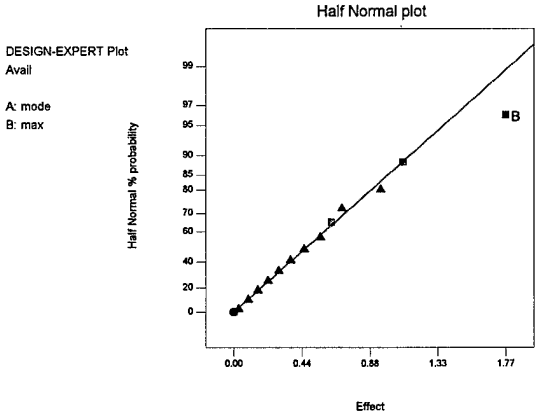
The results in the less extreme cases of incorrectly specifying the triangle parameters indicate potential benefits of replacing the Weibull failure distribution. Smaller absolute errors provide adequate estimates for system performance. The original triangle distribution (with min equal to zero, mode equal to Weibull mode, and max equal to Weibull 98^{th} percentile) is also simulated and returned a relative error of –1.79, -5.93, and –2.32 percent for all three systems (five, ten and fifteen components).

SIGNIFICANCE OF RESULTS

The results of the simulations are analyzed using Design-Expert™ software to determine if the relative errors are significant. The results are a 2^2 designed experiment and the factors (mode and maximum triangle parameters) can be checked for significance. The experiment is replicated three times using different random number streams. The center points are also simulated three times to support the interpretation of the results.

As expected, Design-Expert™ confirms the significant difference in simulated results when replacing the true Weibull distribution with the triangle failure distribution. The relative error for system availability is primarily influenced by the specification of the maximum parameter. The mode becomes significant as the deviations increase to +/- 20% and higher. These results are demonstrated by the two graphs shown below:

10 Comps, +/- 5%



10 Comps, +/- 20%

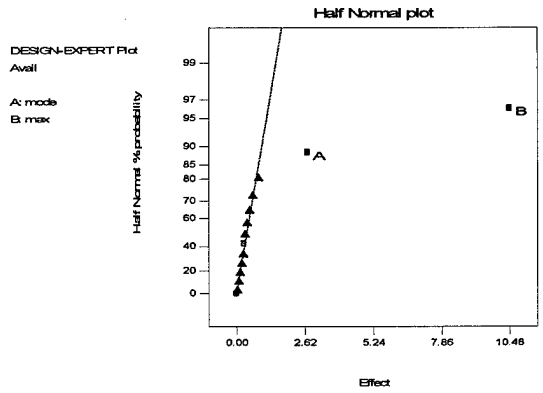
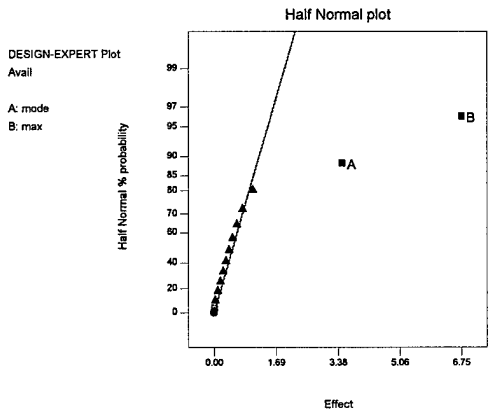


FIGURE 4-4. Normal Probability Plots for Availability

Although the absolute errors are not as excessive in the case of availability, they are still significantly different than when using the Weibull failure distribution.

Similar results are returned when analyzing the Mean Time To First Failure. However, both factors (mode and maximum) are influential in determining the relative error in the MTTFF. The two graphs below indicate the factor significance using the normal probability plot for the effects.

15 Comps, +/- 5%



15 Comps, +/- 20%

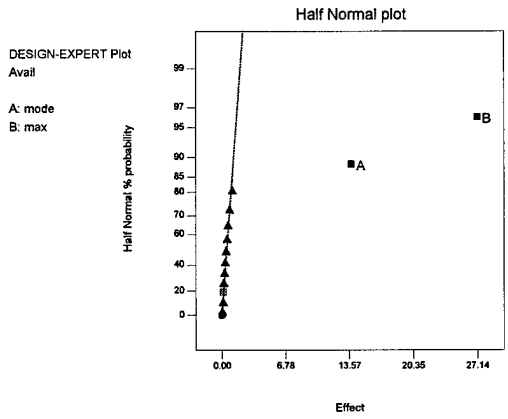


FIGURE 4-5. Normal Probability Plots for Mean Time to First Failure.

CONCLUSIONS – OVERALL FINDINGS

These results support supplementing the true unknown Weibull failure distribution with an estimated triangle distribution to determine complex system reliability behavior. The availability of the system is less affected than the mean time to first failure, but both performance metrics can be approximated using a simpler, easier to understand triangle distribution. This allows engineering judgement to determine the parameters without detailed analysis of fitting a complex Weibull distribution. The triangle is anchored using a zero minimum value and the mode and maximum can be estimated using the engineers' opinion for the most likely and best case scenarios.

The RAPTOR™ simulation software allows easy manipulation of failure distributions, changing from Weibull to triangle in a point-and-click fashion. The availability and MTTFF performance metrics are automatically output and can be analyzed to determine the overall significance of the results. The maximum parameter of

the triangle distribution appears to have more affect on the output of the simulation, however, the mode and interaction should not be ignored. When evaluating MTTFF, the best results are found by underestimating the maximum (98^{th} percentile) value and overestimating the mode. Overall, to get an approximation of system availability and mean time to failure, this research demonstrates the potential of using the triangle distribution to represent component failure behavior.

FURTHER STUDIES

Further studies are recommended in specifying the maximum parameter value of the triangle distribution. The 98^{th} percentile could be changed to converge more closely to the true system behavior when using the Weibull failure distribution. This is especially true when analyzing the mean time to first failure. The 98^{th} percentile, or maximum value should be underestimated to achieve better results. Similarly, future studies could manipulate the mode by including an increase factor to achieve better results. Our research related the 98^{th} percentile to the best case scenario for component failures and allowed an easy approximation of the triangle parameter for the practicing engineer. Additionally, analyzing the sensitivity of output results to incorrect specifications of the Weibull parameters would provide insight into the system behavior. Weibull parameters can be difficult to approximate and this analysis could further support using a simple distribution, such as the triangle. Finally, the entire study could be extended to vary the mode and maximum values by 40% and 50%. The availability analysis indicates less sensitivity and increasing the input variation may provide further support in this research.

Comparing Sparing Strategies

INTRODUCTION

Communications networks must be highly reliable in order to achieve the desired availability for system performance. Once the availability is achieved, maintenance practices drive the underlying cost of keeping the system in operating condition. There are many alternative maintenance concepts that could keep the availability at a predetermined level, yet reducing the annual operating cost. This paper addresses the analysis of a real-world communication network and choosing a preferred sparing structure to minimize annual operating cost. The network is modeled using a generic simulation software for evaluating reliability, maintainability and availability (RM&A).

The communications industry continues to grow, as does the competition among individual companies. Simulation can offer additional insight into network reliability and provide management with decision making tools to direct developing projects. Before a communication network is built and placed in operation, simulation can predict the system performance based on availability. The simulation output exposes areas of the network needing increased attention, and also points out areas less sensitive to changes in the designed structure. Any modifications to the system during this research and development stage can save time and money as the network is developed and put into place. Landers, et al demonstrate using reliability simulation during the design phase; however, their study uses an aircraft communications study.

SYSTEM DESCRIPTION AND MODELING

The communications network is comprised on nine subsystems, spread throughout the country. All components are independent and identical and experience the same rate of failure. The failure rate is exponential with a ten-year reliability of 50%, this implies a mean time to failure of 173 months, or just over 14.4 years, found by using $R(t) := e^{-\lambda t}$ and solving for lamda. The entire network shares a pooled spare and the replacement components are also identical to the original subsystem. There is one replacement unit placed in the spare pool at the beginning of the system operation.

This system is modeled using a modification of the reliability simulation program developed by the United States Air Force. RAPTOR (Rapid Availability Prototyping for Testing Operational Readiness) was developed by the Air Force Operational Test and Evaluation Center in Albuquerque, New Mexico. The simulation package offers a generic point-and-click modeling program to evaluate system performance based on RM&A performance metrics. This software is free to the public and can be acquired by contacting Ken Murphy via e-mail at murphyk@afotec.af.mil. Changes were made to the software and will be described during the explanation of analysis included in this study.

Each of the nine subsystems are treated as individual components and the reliability block diagram shows nine units in series, all sharing a common spare pool. All nine units must be operational in order for the system to be in an up, or operational condition. The spare pool is defined under the maintenance menu when defining the communications block in RAPTOR. The information required by the software includes the name of the spare pool, the initial stock level, the scheduled arrival of new spares, and

the emergency arrival of new spares. For this communication network, there is one initial spare, no scheduled arrivals, and the emergency spare arrival time will change for different maintenance scenarios.

CHANGES TO THE RAPTOR SIMULATION CODE

The RAPTOR simulation code changes affect the robustness of the maintenance declaration module mentioned above. The new coding structure allows the user to define when the emergency spares are ordered and the number of sparing units requested. Previously, only one emergency spare could be ordered and was not requested until a unit failed and no spare replacement was available. Therefore, the system was guaranteed to experience down time associated with the ordering of the emergency spare and replacing the failed component once this spare arrived.

By altering the simulation algorithm, the user can order an emergency spare upon depletion of the spare pool and possibly reduce the down time when the next component fails. If the emergency spare arrives before another unit fails, the system only experiences down time related to replacing the failed unit. The user can also order multiple units to replenish the spare pool. Many times ordering two emergency spares in more cost efficient than ordering them one at a time. The shipping cost can be spread over both spares and the system will experience less down time due to a higher inventory in the spare pool.

INITIAL SYSTEM ANALYSIS

The purpose of the initial system analysis is to find alternative strategies which can support the desired availability performance for the entire network. The main source

of flexibility comes from the increased robustness in the simulation code for handling spare pools. This aspect of maintaining the system is manipulated and the output parameters are compared to evaluating competing strategies. Initially, three input criteria are varied and system availability is used as the metric for comparison.

The number of initial spares is changed from one to two. The ordering policy for emergency spares is changed. In one case, the emergency spares are ordered when a failed unit finds the spare pool empty. This policy is considered "upon request", as emergency spares are not ordered until the are requested by a newly failed unit. The second case orders emergency spares "upon depletion" of the spare pool, i.e., when the last emergency spare is used and before the next unit fails. Finally, the number of emergency spares ordered is also incremented from one to two units and the system availability is predicted. During the first phase of analysis, "upon request" and "upon depletion" spares take one month to arrive.

The simulation tracks the availability for a fifteen-year period. All nine subsystems, or components must be operating for the system to be available. The emergency spares take 30 days to arrive, approximately 1 month. The emergency arrival time is set to 1 month when entering the input data into RAPTOR. Once the spare has arrived the repair distribution is uniform real, ranging from 15 to 30 days. The repair distribution must be specified in terms of months, similar to all other input into the RAPTOR software for this example. The unit of time needs to remain constant. Therefore, the repair distribution is specified as uniform real, with the parameter .5 to 1 month.

The table given below summarizes the initial system analysis. The purpose of this phase is to gain an understanding for the system and to determine potential avenues for increasing the availability while reducing overall cost. This is the first step in providing a solution to decision makers on which direction to follow in maintain this communication network once the system is fully operational.

Number of Initial Spares	Ordering Policy	Number of Spared Ordered	System Availability
1	Upon request	1	92.0%
1	Upon request	2	93.8%
2	Upon request	1	92.6%
2	Upon request	2	94.1%
1	Upon depletion	1	96.1%
1	Upon depletion	2	96.2%
2	Upon depletion	1	96.1%
2	Upon depletion	2	96.2%

TABLE 4-5. System Availability with Changing Replacement Policies

Currently, the preferred strategy is to make one spare available at the onset of network activation and follow this up with emergency spares only upon request. This only provides a 92.0% system availability. This value is slightly lower than the desired 93% availability for the operational system. Ninety-three percent is the assumed target

value for this communication network. Changing to an ordering strategy upon depletion of the emergency spares shows the greatest signs of improving the system performance, increasing the availability to 96.2%. After this jump, other changes provide minimal increase to the availability and will not be considered in further studies.

New areas of focus derived from this initial step in analysis. Ordering upon request guarantees additional down time while the systems waits in a failed condition until the emergency spare arrives. Therefore, shortening this down time would increase availability. The emergency arrival rate is adjusted in the next phase of analysis to determine the level of impact on availability. Also, the length of time experience before the first failure is considered as a means of applying schedule maintenance, or spare ordering to increase availability. One way to select a re-order period may be to examine the availability over time to see when this value drops below the desired level.

SECOND LEVEL ANALYSIS

The first area to consider in this step of the analysis is the ordering time. The emergency spare arrival time currently takes 30 days. The simulation model is manipulated to vary this parameter from 10 to 60 days for "upon request" sparing and 10 to 180 days for "upon depletion" sparing. The analysis compares both ordering policies when re-running the simulations. This demonstrates the affect of order time on both policies and may offer a cost saving in both directions. Shortening the order time will increase the "order upon request" availability, while lengthening the order time may not decrease the "order upon depletion" strategy to a great extent. Both of these scenarios are important to examine.

The graph below plots availability versus ordering time, with both strategies plotted on the same chart.

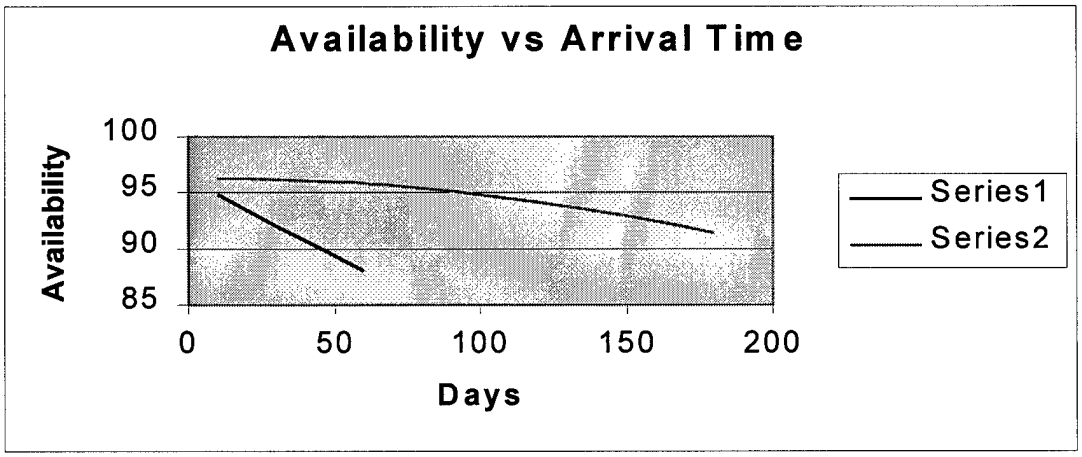


FIGURE 4-6. Availability versus Re-order Time

Series 1 represents ordering "upon request" and series 2 is ordering upon "depletion" of the last emergency spare. This chart clearly shows the increase in availability as the ordering time decreases. The benefit is far greater when ordering "upon request" as expected. This directly decreases the amount of down time experienced by the system. When ordering "upon depletion", changing the order time is not as dramatic since the system may or may not experience down time before the replenishment spare arrives. The availability is practically constant with a re-order time anywhere from 10 to 60 days using this policy. This does provide valuable information and analysis for alternative strategy determination.

This analysis leads to competing strategies offering comparable system availability. Alternative 1 is maintaining the system with a 20 day re-order time and an "upon request" ordering strategy for emergency spares. This provides system availability approximately equal to 93%. Alternative 2 is using "upon depletion" ordering strategy and allowing the order time to increase to 150 days, or 5 months. Although this order time seems high, the system availability is near 93%. Both these strategies are compared on the basis of cost in the final step of this analysis.

The other area of focus for phase two of this scenario deals with scheduled maintenance, or predetermined intervals for ordering emergency spares. Two graphs are given below to aid this analysis. The first graph is a histogram showing the mean time to first request for an emergency spare. One component (of nine) in the system fails, uses the initial spare and then at the time of the second component failure the simulation run time is captured. This graph indicates a possible re-order interval for scheduling the first emergency spare arrival. The second graph is the system availability over time. The system is simulated over 4 months and the graph shows when the availability drops below our assumed goal of 93%. This also adds information to help determine a scheduled re-order period.

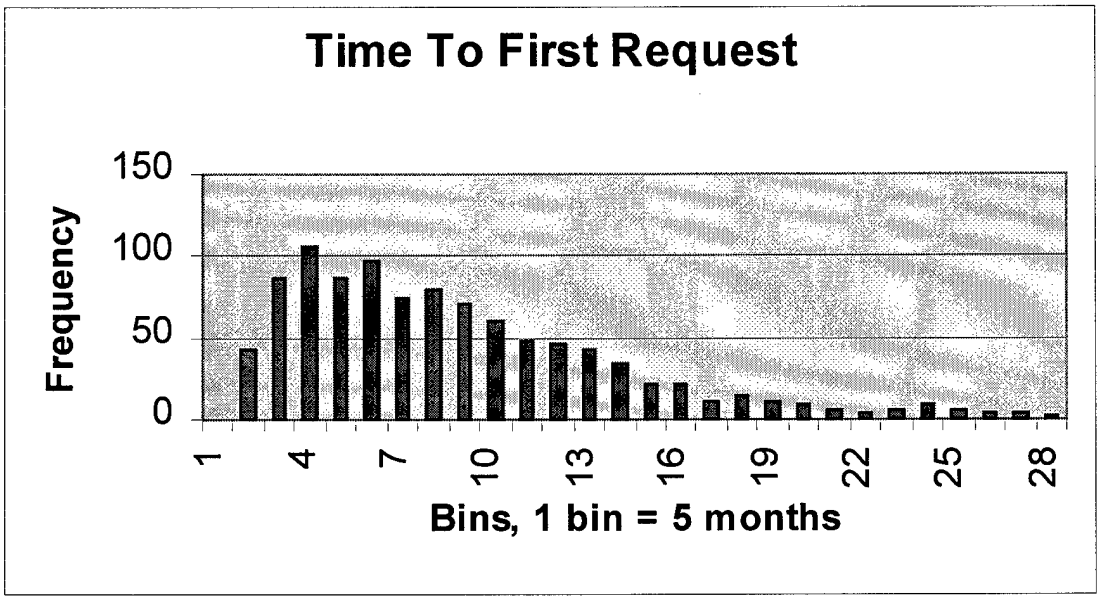


FIGURE 4-7. Mean Time to First Request

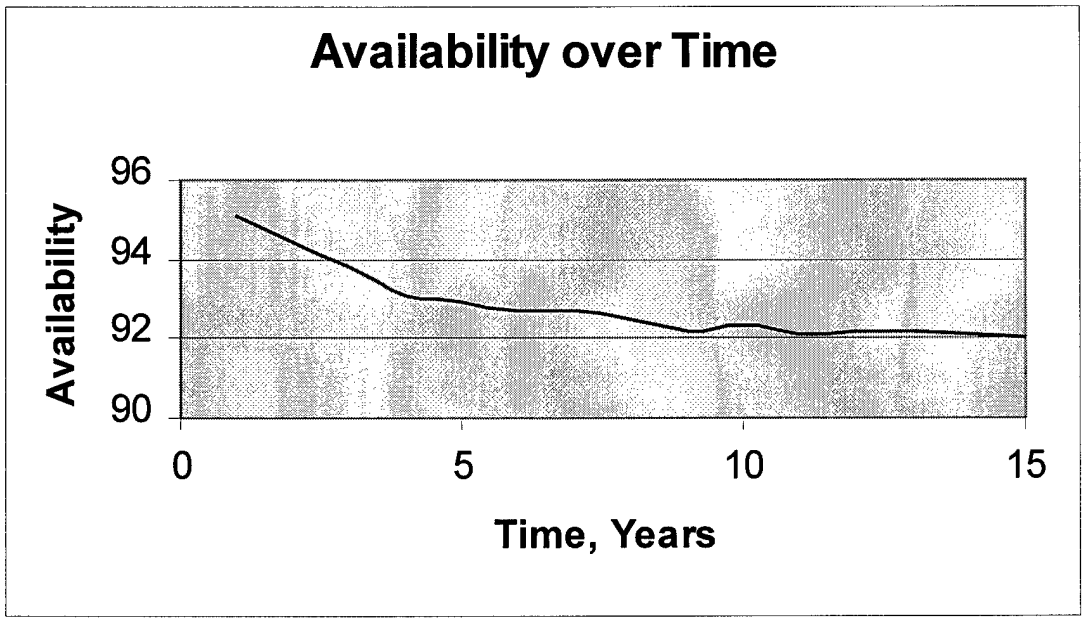


FIGURE 4-8. Availability Over Time

Examining these graphs gives some indications that a scheduled re-order period for emergency spares could improve the system availability and offer a third alternative for sparing policy. The mean time to first request for emergency spares is just over 40 months, close to 3 ½ years. The system availability drops below 93% around the 5-year point. By ordering spares on a schedule interval, the availability may increase or stabilize near 93%. The simulation input is modified and scheduled arrivals are added to the emergency spare pool. The table below summarizes new simulation runs features the addition of scheduled spare arrivals. The arrival interval is changed in increments of 12 months or 1 year.

Arrival Interval	Availability	Ending # of Spares	Unscheduled Spares
No sched arrivals	92.0%	0	8.128
1 year	96.0%	6.19	0.45
2 years	94.9%	1.42	2.64
3 years	94.0%	0.36	4.46
4 years	93.5%	0.23	5.35
5 years	93.1%	0.07	6.15

TABLE 4-6. Availability Results for Scheduled Arrive of Spares

This table clearly indicates an advantage to scheduling spare arrivals. The disadvantage lies in the number of unused spares remaining at the end of the 15-year

period. As the frequency of the arrivals increase, the availability increases and so does the unused number of spares, as expected. The number of unscheduled spares signify the number of failures still requiring an emergency spare due to a depleted spare pool. The emergency spares take 30 days to arrive. Uses 93% as our assumed goal, a scheduled order period of one spare every 5 years seems appropriate. This leaves of basically no spares left after fifteen years and requires 6 additional spares to be ordered under emergency situations. A spare is not ordered at the fifteen-year point, only at time 5 and 10 years. This is the Alternative 3 to be used for cost comparison in the final section of analysis. Eliminating the third scheduled spare arrival eliminates the excess spares in inventory at the end of the operating period.

FINAL ANALYSIS AND COST COMPARISONS

A brief review of the three alternatives provides an overview for the competing strategies in minimizing the cost of maintaining a 93% availability for this communications network. All three alternatives have one initial spare and the ability to order emergency spares if the spare pool is empty. Alternative 1 orders emergency spares only when the next component fails and finds the spare pool empty. This is considered "upon request" and the spare takes 20 days to arrive. Alternative 2 orders an emergency spare when a failed unit uses the last spare in the pool. This is considered "upon depletion" and the spare takes 5 months to arrive. The final strategy is a mix of scheduled and unscheduled ordering. A scheduled spare is ordered at the 5^{th} and 10^{th} year and emergency spares are ordered when a failed component finds the spare pool empty and takes 30 days to arrive. These three alternatives are considered equal on the

grounds of availability (all approximately 93%: 93.4%, 92.9%, 93.1%) and are evaluated now on the basis of cost.

The three alternatives require 9 spares for the 15-year simulation. One initial spare and eight back-ups, ordered based on the given strategy per alternative. Therefore, the sparing cost is not considered and the comparison is done based on order policy and the time value of money. The ordering policy effects when the spares are ordered over the lifetime of the communication network. The "upon request" policy delays ordering until the next unit fails while "upon depletion" orders an emergency spare sooner to prevent guaranteed down time with the next failure.

The priority placed on the filling the spare pool also affects the cost parameters. A scheduled replacement is less expensive than a spare required within 20 days. The maintenance cost varies with the time allowed to fill the spare pool. Specifically, alternative 2 takes advantage of this due to the long (5-month) ordering window. Alternative 1 requires the fastest delivery of the sparing unit, only 20 days. Alternative 3 includes both scheduled spare arrivals (less expensive) and emergency spare arrivals within the 30 days. Since alternative 3 requires the both options simultaneously, the 30-day delivery is assumed to cost the same as the 20-day delivery. The ratio of scheduled delivery (every 5 years), to "upon depletion" (5 months) delivery, to "upon request" (20/30 days) delivery affects the final cost of maintaining the availability of the network.

RAPTOR simulation provides a number of output statistics and data files to build a valid cost equation for each of the alternatives. Since the time value of money is considered an influence to the problem, the simulation must report when the spare

ordering takes place. Each alternative is modeled and simulated, building a histogram of time to order for each spare replenishment activity. Similar to the graph depicting time to first request, RAPTOR provides the data necessary to characterize the time to second request, third request and so on. The data is truncated at 15 years, since no spares are order after this point. In alternative 2, the data is truncated after 14years and 7 months due to the 5-month ordering time. The cost is strictly based on a 15-year availability for the communications network.

Alternative 3 requires additional output from the simulation software. The number of emergency spares is tracked and used to augment the spare pool in addition to the scheduled arrivals. The simulation is executed for every 1-year period and the cost is based on ordering the emergency spares at the end of that period. Using the beginning of the 1-year period as the order time would penalize alternative 3 based on the time value of money. The simulation output reports cumulative number of delays to the system. A delay represents the need for an emergency spare. A routine failure with a spare in the pool does contribute to down time of the system, but does not signify a "delay" in terms of the statistics collected by the software. Delays strictly refer to halted operation due to the depletion of the spare pool. Therefore, the cumulative delays are subtracted from each previous value to determine the number of delays in each 1-year period and the number of "upon request" orders experienced.

All this information is used to calculate the cost for each alternative and then compared to choose the best ordering policy. In this study, a 10% return on investment is assumed. The cost for scheduled spare arrivals is fixed at 1 and the ratio of the other two

ordering policies vary from 1 to 10. As an example, a ratio of 1:2:4 implies a cost of 1 to replenish spares on a scheduled basis, 2 for "upon depletion" taking 5 months, 4 for "upon request" taking only 20/30 days (30 days for alternative 3 as discussed earlier). The rate of return is not as dramatic a factor when dealing in monthly availability, however, this factor is included as the problem could be extrapolated to system availability over years.

The chart below indicates the alternative selection based strictly on the least cost. There are other advantages and disadvantages to each alternative and they are discussed at a later time.

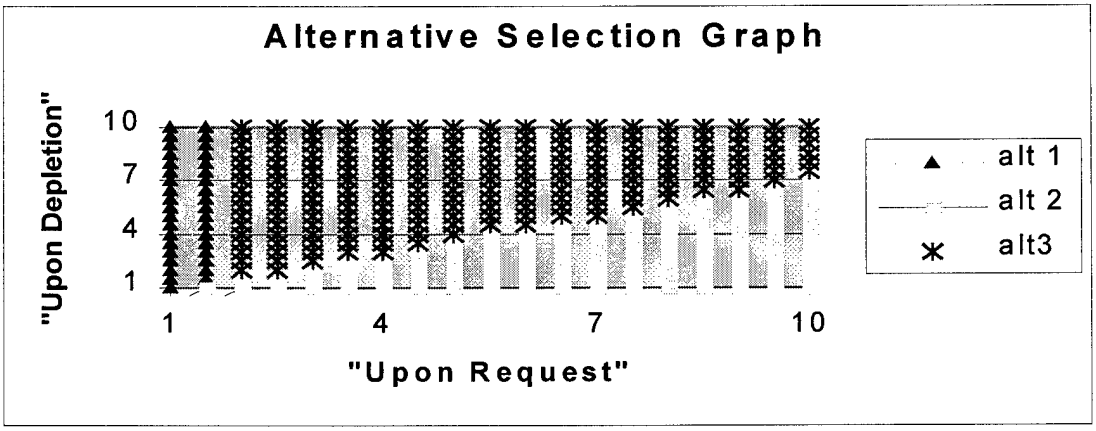


FIGURE 4-9. Alternative Selection Graph

Each series represents an alternative ordering policy. Alternative 1 dominates until "upon request" ordering is twice that of scheduled ordering. After this point, the alternative 2 and 3 compete against one another. The new ration of importance is

between "upon request" versus "upon depletion". The more expensive "upon request" is compared to "upon depletion", the more likely alternative two is selected. Alternative 3 is always chosen when "upon depletion" is more expensive than "upon request". The costs are approximately equal when the ratio is 3:5, "upon depletion" related to the cost of "upon request". (This is seen in the slope of the boundary line dividing these two alternatives.)

The cost equations are examined to determine their two-way interactions or boundaries. By setting the ratio equal to 1:1:1 for scheduled, "upon depletion" and "upon request", the boundary conditions are clarifies. Alternative 1 and 3 vary due to the change of the "upon request" value, the scheduled cost is always set equal to one. Alternative 2 varies due to the cost of the "upon depletion" ordering. The cost equation is equal to the sum of the failure times included in the present value equation. PV = Av^n, where PV is present value, A is the value amount, v is the ratio 1/1+roi (return on investment, the interest rate), and n is the number of time period into the future. The summing variable from the simulation is the n variable and is found by setting A equal to one, i.e., the 1:1:1 ratio.

The equation to compare alternatives 1 and 3, and 1 and 2 are given below:

$3.64 * C = 3.08 * C + 1.01$ (alt 1 = alt 3)

$3.64 * C = 4.38 * B$ (alt1 = alt 2)

The value of C is the value of "upon request" ordering. The 1.01 constant value is due to the scheduled arrivals in alternative 3. The value of B is the value of the "upon depletion" ordering. Solving C in equation 1 identifies the domination point of

alternative 3. Once the "upon request" ordering cost is greater than 1.80, alternative 3 is the dominant choice. Below that value, the ratio of B/C from the second equation is .831 and dictates whether alternative 1 or 2 is less expensive. If the ratio is greater than .831 than alternative 1 is chosen, otherwise, alternative 2 is chosen. This is verified by the graph of alternatives given above. With "upon request" equal to 1.5 and "upon depletion" equal to 1.0, alternative 2 is chosen as the least expensive because 1/1.5 = .67 which is less than .831. As "upon depletion increased increasing the ratio above .831, alternative 1 became the dominant solution. Keeping in mind, alternative 1 is dominated once the "upon request" cost exceeds 1.80. After this point, the tradeoff is between alternatives 2 and 3.

Alternatives 2 and 3 are dependent on the cost of "upon request", "upon depletion", and scheduled ordering (remains fixed at 1). The equation is given below describing this relationship:

$$4.38 * B = 3.08 * C + 1.01 \quad \text{(alt 2 = alt 3)}$$

$$B = .703\ C + .231 \quad \text{(equation of dividing line)}$$

As expected, the relationship between alternatives is not a constant value due to the fixed cost from the scheduled ordering in alternative 3. The ratio of B to C converges to .703 as the "upon request" and "upon depletion" cost increase. This equation of the dividing line is the boundary line between alternatives 2 and 3 and is highlighted in the cost comparison graph. As "upon depletion" cost rise in relationship to "upon request" cost, alternative 3 becomes the dominant solution. Overall, these fixed boundaries between

alternatives 1, 2, and 3 can be found once the simulation determines the re-ordering times (the variable n in the present value equation).

CONCLUSIONS AND RECOMMENDATIONS

The final alternative selection rests in the minimum cost of maintaining the ordering policy. The are significant differences between ordering spares for arrival in 20-30 days versus 5 months. An "upon request" ordering policy requires additional manpower, expensive shipping cost, and expensive inventory of communication spares. "Upon depletion" ordering provides flexibility and a larger window for spare arrival. The scheduled spares are assumed the least expensive and provide some benefit to maintaining a high availability while reducing the number of emergency, or "upon request" orders.

The present value of money is an equalizing metric to compare the alternatives. By waiting to order spares until required, alternatives 1 and 3 take advantage of a return on investment of unused capital. These alternatives also benefit from smaller inventory in the spare pool since the requested spare is put into operation upon its arrival, inventory is kept at zero. Alternative 3 does maintain inventory in the spare pool due to the scheduled arrivals, but not when ordering emergency spares. Alternative 2 maintains inventory in the spare pool as replenishment occurs. This happens each time the inventory is depleted. This increases the number of unused spares at the end of the simulation but reduces the amount of down time. Assuming a premium for high availability, alternative 2 may become a dominant solution.

Overall, the graph for alternative selection guides the user's selection based on the minimum cost. The graph is dictated by the ordering policy costs, and the ratio among those values. Analytical boundaries are found between the alternative by using the simulated ordering time and calculating the present value of each alternative. The goal of the simulation is to provide flexibility for defining the maintenance strategy for achieving high availability of the communications network. As input parameters change, the generic simulation is modified to provide new output to generate new cost comparisons. This analyzes can also be used to drive maintenance costs to a certain value. If management is focused on using a given alternative, the graph indicates where this alternative is least expensive. The user must focus on reducing the ratio of one ordering cost to another to ensure they fall in the appropriate area of the graph.

Time Between Failure (TBF) vs Time To Failure (TTF)

INTRODUCTION

The primary focus of this article is to analyze the outcome of misunderstanding the time between failure (TBF) and the time to failure (TTF), often called the time to first failure (TTFF). The TBF and TTF deserve additional attention and explanation due to the confusion in reliability and system analysis. The time between failure distribution of a complex system is exponential (Barlow and Proschan (1965)). Regardless of the individual component's time to failure densities, the overall system TBF can be described with an exponential distribution. The same can not be said about the system time to failure. The TTF is not exponentially distributed and the practitioner errs when misunderstanding this result.

The application of Barlow and Proschan's (1965) exponential TBF can be detrimental to system analysis. Usher (1993) demonstrates misapplication of these results in his case study. Analyst applying this concept to the TTF and modeling the system with an exponential distribution cause unneeded sparing requirements and drive the cost of maintaining the system to unjustly, high values. There are additional circumstances increasing the confusion of this error. Not only must the proper distribution be selected, but the proper parameters must be determined as well. The error in using the incorrect exponential distribution is magnified if the practitioner also uses the wrong parameters. All these errors are explained and demonstrated in the given case study.

The intent of the case study is to model a single system and then project these results into a scenario of multiple systems. The single system is simulated based on component information to determine the mean time to failure and mean time between failure. These distributions are then extrapolated into a higher level system requiring multiple units. The overall system is simulated to determine sparing requirements and calculate cost. The distributions driving the systems are modified to demonstrate the change in sparing and cost as the wrong distributions are used. All simulations are done using a generic reliability software package called RAPTOR. Background on this program is given in later discussions.

EXPLANATION OF THE SYSTEM

The system used for experimentation in this case study in an automobile. The reliability block diagram (RBD) for this vehicle is given below, along with a table of the

component failure distributions. The RBD is simplified and the distributions are based on days as the unit of time. This scenario is somewhat simplified but the intent is to demonstrate the effect of misinterpreting the TBF/TTF fundamentals.

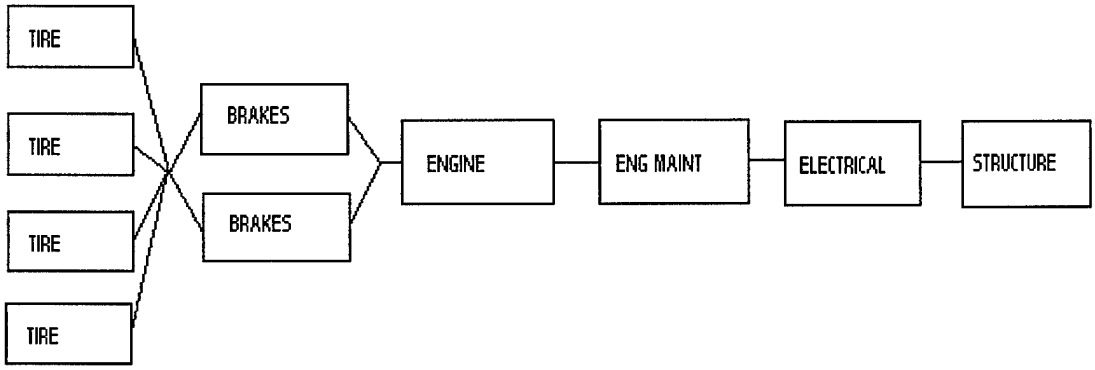


FIGURE 4-10. Reliability Block Diagram

Component	Failure Distribution	Mean	Repair Distribution
Tires	Normal(365,36.5)	365 days	Lognormal(5,.5)
Brakes	Weibull(2.0, 540)	479 days	Lognormal(10,1)
Engine	Lognormal(275,45)	275 days	Lognormal(30,10)
Engine Maintenance	Weibull(3,300)	268 days	Lognormal(4,.4)
Electrical	Triangle(100,500,1500)	700 days	Lognormal(21,2)
Structure	Gamma(3,175)	525 days	Lognormal(21,5)

TABLE 4-7. Failure and Repair Distributions for Individual Components

This system is then rolled into a fleet of vehicles in which each car is a separate system. There are fifteen vehicles, twelve of which must be operating in order for the

fleet to be considered operational. This allows the availability of the system to be tracked during the simulation. The fleet RBD is simply a 12/15 parallel network with identical subsystems (the cars).

INITIAL SIMULATIONS

RAPTOR simulation software is a generic modeling program developed by the United States Air Force and available at no charge by contacting murphyk@afotec.af.mil. The Air Forces uses RAPTOR to provide reliability, maintainability and availability (RM&A) analysis on new equipment acquisitions. Many times, the actual equipment can not be testes or tested long enough to verify system requirements. Simulations using RAPTOR provide answers to RM&A questions and predict system parameters at a reduced cost. The software is generic in nature and the user develops RBD's to analyze using a point-and-click, menu-driven approach.

Modifications to RAPTOR were made as part of ongoing research into advancing RM&A analysis. The changes focused on adding cost as input parameters to the simulation environment. The total cost for operating and maintaining a system is output at the end of each simulation run. The overall cost can be used as an output parameter to differentiate among modeling alternatives. In this case, cost is used to highlight the error of misunderstanding the TBF and TTF distributions. The sparing output is already included as a feature in the original RAPTOR software. The sparing requirements and availability of the systems are the other parameters used in this case study.

The vehicle RBD is constructed in RAPTOR and defined by the appropriate time to failure and repair distributions. RAPTOR models the systems and outputs the

individual failure times. Initially, the system is modeled until the first failure and this is repeated 1000 times. The ending simulation times are output to a text file for later analysis and signify the time to first failures of the vehicle. These data value are used to construct the TTF distribution. Next, the system is simulated for 2000 days to determine the number of system failures. There are 29 systems failures, thus, the system must be simulated for approximately 70000 days to generate the 1000 failures. The time between each failure is captured by RAPTOR and output to a text file. These values describe the TBF distribution for the system.

The data files containing the failure time values for both the TBF and TTF are analyzed using Best Fit software. The histogram for each distribution is shown below for the 1000 data points.

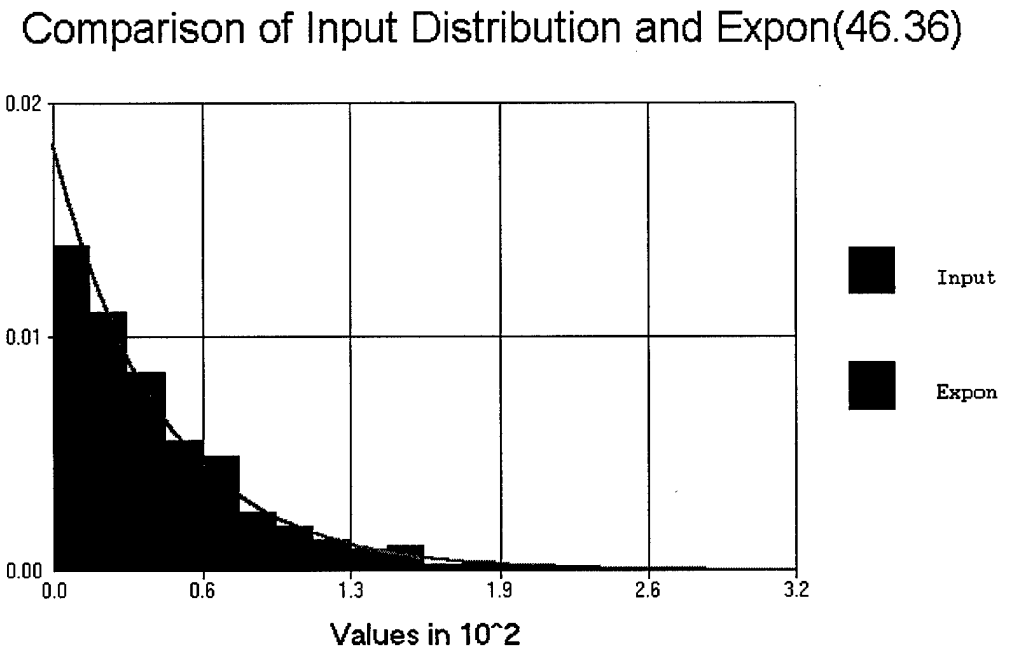


FIGURE 4-11. Time Between Failure Distribution

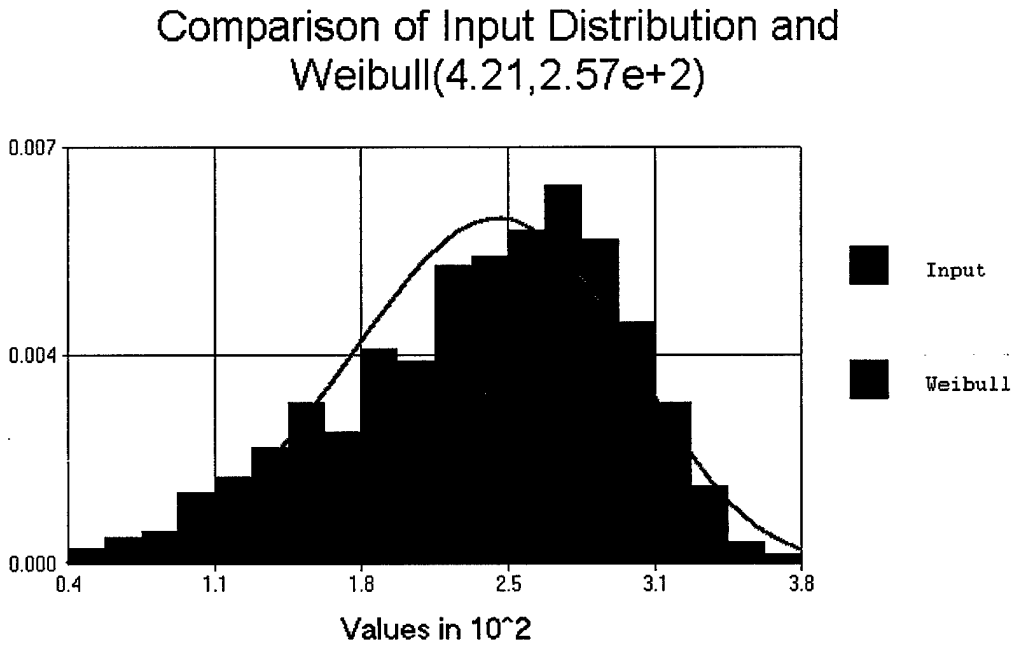


FIGURE 4-11. Time to Failure Distribution

These two graphs highlight the differences between time between failure (TBF) and time to failure (TTF). The first graph is clearly exponential distributed and passes the chi-square test. The test statistic is 21.6 and the chi-square critical value with 19 degrees of freedom with a 10% alpha value is 27.2. Therefore, the null hypothesis that this data is from an exponential distribution can not be rejected. The second graph is clearly NOT exponential. These are the TTF values and a Weibull distribution is plotted over the histogram. This distribution is used in later analysis to describe the vehicle TTF and determine the true sparing requirements and cost evaluation. The Weibull distribution is not an exact fit but represents a reasonable interpretation of the data.

EXTRAPOLATING VEHICLE RESULTS TO A FLEET

The next stage in the scenario is to extrapolate these results into the fleet. The fleet of fifteen vehicles is simulated using the distribution data collected from the individual automobile. The incorrect and correct data is modeled and the results are collected based on sparing requirements and cost. The cost of a vehicle is assumed to be $1,000. The penalty for not meeting the required availability (12 of 15 vehicles operational) is assumed to be $1,000 per day. This represents lost revenue, customer disappointment, and lost market share. The fleet is simulated for a 2-year period and the data collected is used to determine resources allocation for a 10-year period. These statistics are based on 50 runs from the RAPTOR simulation.

The three simulations use an exponential time to failure for the individual auto in the fleet setting. This represents the incorrect assumptions and misinterpretation from the Barlow and Proschan (1965) results. The mean is equal to the mean from the TBF

acquired in the initial simulations. Next, the incorrect exponential is used but the correct mean from the TTF data is input as the distribution parameters. Finally, the correct TTF distribution is used to describe the individual vehicle's failure rate and the Weibull distribution represents this characterization. The distribution does not exactly fit the data but is a very close approximation as indicated by the graph provided.

The table given below summarizes the scenarios described above. In each case, the sparing and cost information is collected from the simulation output and is discussed in further detail following the tabulated results.

Vehicle Failure Distribution	Availability Over 2 yrs	Cost due to lost availability	Sparing Requirements	Cost due to sparing reqts
Expo(46.4)	50.1%	$360,000	176	$176,000
Expo(233)	99.3%	$5,000	43	$43,000
Weib(4.21,257)	99.0%	$8,000	37	$37,400

TABLE 4-8. Cost for Alternative Sparing Strategies

The results of this scenario are magnified when the user multiplies the tabulated data by five in order to project spares and costing over a 10-yr period. The RAPTOR simulation software provides all the data in one form to understand the overall system performance.

SIMULATION RESULTS

The most troublesome error is using the mean time between failure distribution in place of the mean to time failure distribution from the vehicle simulation. Using the data in the fleet model cause 375% increase in sparing allocation, moving the required spares in a ten-year period from 880 instead of the realistic 185. The sparing is grossly overestimated and results in a tremendous surplus of vehicle spares, or more importantly, allocated resources to purchase these spares. The amount of capital tied up in expected sparing requirements results in poor management of company resources and capital allocation.

The secondary, and much less damaging error, is to assume the exponential distribution and use the time to failure statistics. This scenario models the fleet with an exponential distribution with a mean of 233 days versus the realistic Weibull distribution with the same mean. This causes additional sparing requirements due to the increased variability of the distribution. The sparing increases by 16% from 37 to 43 over a two-year period. Multiplying the cost factors by five to allocate capital over a ten-year period the amount of resources increase from $187,000 to 215,000. The availability is slightly higher in this case due to an increased number of simulation experiencing no failures. The cost due to availability actually decreases from $40,000 to $25,000. Therefore, the overall cost increases from $227,000 TO $240,000.

The key parameter to examine is sparing allocation. The Weibull distribution best fits the TTF data from the vehicle simulations and should be used to describe automobiles in the fleet modeling. The fleet modeling evaluation criteria may change, but the sparing

requirements are still going to drive the system and cause erroneous results when the wrong distributions and parameters are used.

CONCLUSIONS AND RECOMMENDATIONS

An important angle to consider sparing allocation errors is in terms of inventory and capital expenditure. The situation must be examined as applied to other systems and scenarios, other than the automotive/fleet example provided. Many companies build inventories based on sparing requirements. Once the lifetime of the system has ended, i.e., the ten-year period, the system may be obsolete. Now the company maintains an extreme overage of sparing inventory and must dispose of the unused spares. The cost of acquiring the initial inventory was excessive, along with storing the inventory over the life of the system.

The importance of understanding the difference between TBF and TTF can not be iterated enough. The eventually outcome of misusing these distributions negatively affect a companies ability to properly allocate resources and plan for accurate system behavior. As the availability metric highlighted in this example, a company is prepared for 50% down time, when in fact, experiences less than 2% down time. This problem becomes more detrimental during design and development. An experimental system may be ignored or completely re-designed due to an error in modeling and performance prediction. A company may over design and thus expend additional resources to increase an erroneously low availability metric.

All these errors affect management decision making and demonstrate the flaws of misinterpreting Barlow and Proschan (1965) results concerning the exponential TBF.

The results of this example support the exponential time to failure, but contradict these results when the distribution of choice is the time to failure. These two distribution can not be confused. Kececioglu (1991) recommends using mean time to first failure (MTTFF) to further clarify the difference in distributions. The mean time between failure (MTBF) is exponentially distributed for complex systems with multiple time to failure densities. The MTTFF is Weibull in shape and significantly impacts future modeling and analysis based on this distribution.

Summary

The triangle distribution study demonstrates the potential of using reliability simulation early in the design and development phase. Often times, little or no data is available on new components and a distribution must be assumed. The triangle distribution is easy to estimates and to understand relative to engineering judgement. The results indicate insensitive availability and mean time to first failure performance metrics. An additional case could be analyzed without changing all of the component failure distributions. A certain percentage of the components would maintain the Weibull distribution and only a small number would change to the triangle. This may be more realistic than the worst case scenario presented in this chapter. Additional comments are provided in Chapter 5 under future research.

The second section of this chapter validates real world application of reliability simulation. The communications network and trade off analysis is part of an ongoing research effort at a local company. The modifications to the RAPTOR software were provided to the engineers and the simulation results were briefed. The highlight of this

case study is the structure outlined for completing trade off analysis. The same steps can be used to apply reliability simulation to other situations and offer alternative solutions to decision-makers. The calculation of present value dollars provides a valid metric for alternative selection.

The final section of Chapter 4 focuses on clarifying the two, often confused, distribution. Time between failure and time to failure are misinterpreted and misapplied in the literature and in application. The study highlights the differences in the two distributions and demonstrates the error of implementing the wrong distribution in system evaluations. The logistics considerations are grossly overestimated which can lead to excessive cost expenditures. Applying the proper distribution during the simulation modeling is critical to accurately predicting system performance.

CHAPTER 5

SUMMARY AND CONCLUSIONS

Contributions

The primary focus of this research is to advance the understanding of system-level reliability, maintainability and availability (RM&A) analysis. Reliability software provides an opportunity to develop, test, and demonstrate RM&A insight. The practicing engineer can model system behavior with little understanding of simulation code writing, and software development. The generic nature of the enhanced RAPTOR reliability program removes the barrier to simulating system performance.

Modifying the RAPTOR simulation code is just the beginning to gaining additional insight into RM&A analysis. The increased robustness of the software handles new features, offering the user capabilities previously ignored. Changes to the software are explained and minor case studies are provided to highlight the improvements to the simulation environment. The new capabilities include priority queuing, stream dependency, repair by adjustment only, reliability improvement/degradation, block level definitions, an overall cost structure, and changes to emergency sparing for pools. These modifications stem from user suggestions and increase the flexibility of accurately modeling real-world systems.

The true contributions demonstrate the potential of using this powerful simulation tool. The reliability analyst is often faced with little or no data available to build an appropriate input distribution for component behavior. This research addresses the problem by examining the sensitivity of output metrics to changing input distribution

specifications. Precisely, the triangle distribution is substituted for the Weibull distribution to determine the effect on the system availability. The parameters of the triangle distribution are easily approximated and can be estimated using engineering judgement. The results of this research verify the insensitive nature of system behavior, allowing accurate prediction of performance with insufficient input data. The input parameters for the component's failure distribution are varied as much as 30% with little impact on system availability. Overestimating the triangle mode and underestimating the worst case failure (maximum parameter) provides accurate prediction for the mean time to first failure. The triangle distribution offers a valuable alternative to the Weibull distribution when little or no data is available.

The communications example is provided to validate this research in a real-world case study. The network is derived from an existing system being developed by a local company. The enhance reliability software is used to provide alternative sparing strategy for maintaining a required system availability. Furthermore, cost comparisons are done based on the present value of capital and recommendations are made as to which alternative provides the best sparing solution. The sparing policies are compared using simulation output and the advantages and disadvantages of the strategies are discussed. The outcome from this research directly affected the decision making process involved in this ongoing project. The case study answered questions regarding RM&A issues and an gave the reliability engineer an algorithm to follow for future analysis.

This research continues past the applied problem of the communications network to explain an issue of discrepancy in the reliability literature. The fundamental difference

between time to failure and time between failure is addressed and the implications of erroneously interpreting these distributions are analyzed. Barlow and Proschan (1965) correctly build the exponential distribution from time between failures of a complex system with multiple failure densities. The faulty analysis in the misapplication of these results to system level analysis. The time between failure density is exponential, not the time to failure density. Applying the incorrect density in the analysis of system performance drives the output metrics in the wrong direction. The sparing example is provided to examine the error in using the wrong distribution and the incorrect parameters. The resources allocated to handle sparing and logistics, along with the capital expected to support this system, are greatly overestimated and inaccurate. The faulty analysis leads to an abundance of spare parts and restricted capital that could be better spent in other areas of product development.

Areas of Future Research

The field of reliability continues to grow and so does the opportunity to provide beneficial research to the literature. The areas for future research are two-fold, reliability simulation and RM&A issues in general. Both areas provide ample space to contribute to the literature and provide practicing engineers with additional tools to better complete the job of system analysis. The enhancements to the simulation software are derived from user input and customer feedback. The list of features, however, is endless as the user demands greater flexibility in modeling the real-world system.

The area of preventive maintenance remains an issue with reliability simulation. Specifically, RAPTOR does not provide an option to input preventive maintenance on a

fixed time interval or monitor based schedule. The maintenance is strictly based on failure densities, and the downtime of the system is used to repair a failed component. The idea of preventive maintenance would have to be tied to system status. The component should be maintained when the system is down, or halting the component's operation does not affect the system status. The current simulation does not assess the system status prior to failing an individual component. Adding this ability would increase the maintainability aspect of the simulation environment.

Ongoing research at the Air Force Operational Test and Evaluation Center continues to upgrade the RAPTOR simulation software. New features are currently being added to handle cost parameters, layered definitions, automated graphing capability, and other features. Layered definitions allow the user to click on a pre-defined block and expose the underlying structure used to create that object. The intent is to provide the user the capability to define a subsystem using a reliability block diagram (RBD) and in turn, use that subsystem as a component in a higher level RBD. Automated graphing allows the user to identify warm-up periods, convergence of output parameters, and initial fluctuation due to statistical variance. The upgraded software also addresses many graphical user interface concerns and tries to increase the ease at which new customers can use the software.

An area completely open to improve this research is optimizing system performance based on cost. The idea of incorporating cost as an input and output parameter in the simulation software was introduced in this research. Tying the cost parameters to the design the system being modeled opens a new avenue to improving

reliability analysis. The reliability block diagram could be constructed and the potential components to be involved in optimization would be flagged. The simulation would generate the overall cost of maintaining and operating the system. Afterwards, the reliability of the individually flagged components would be incremented according and a new cost would be calculated. The manner in which the optimization takes place, along with the method for implementing the algorithm into the programming code is an area of continued research. The goal would be to minimize the cost of a given system by optimizing the reliability of individual components. The software would identify the "biggest bang for the buck" as the reliability improvements would be assigned a cost that would be factored into the overall equation. The implementation of this idea would have to involve external modification to the software providing a method for changing the RBD and tracking the system cost for each configuration.

A simple area of research is to consider expansion of the small case studies provide in Chapter 3. These case studies demonstrate the significance of the code changes but could be investigated separately as area of potential research. As an example, the concept of repair versus remove and replace maintenance could be coupled with "bad as old" and "better than new" repair actions to advance system maintainability strategies. The features are present in the enhanced software to facilitate this research. Reliability improvement and degradation continue to be areas of concern for engineers. The simplified case study provide in Chapter 3 demonstrates the utility of incorporating this feature into the software and could be used as a launching point for further analysis.

A specific area of research would be to re-evaluate the triangle distribution, sensitivity study. In this research, 100% of the components' failure rates were changed to the triangle distribution. This is a worst case scenario. Often times we have data on some of the components and must approximately a small number of distributions without available data. In this case, the scenario could be re-evaluated when substituting the triangle distribution for the Weibull distribution on a small percentage of the reliability block diagram components. The percentage could be decreased from 50% to 10% and examine the results of the simulations. The availability and mean time to first failure metrics should be closer to the true system behavior as the percentage decreases. This may be more realistic than the worst case scenario included in this researach.

A final area of research lies in the area of time manipulated simulation. All the changes to RAPTOR were made based on events during the simulation, such as failures and repairs. However, the system may experience changes to the original parameters or design based on the passage of time. A couple of examples may help clarify this situation. Reliability improvement may occur when new components are releases based on a monthly update to new devices. Therefore, every month the mean time to failure would increase and change the reliability of the individual component. The changes made in this research allow the reliability to improve after each failure, regardless of the surpassed time.

Another example deals with aircraft performing a given mission. Assume the scenario is based on a four-engine aircraft. During take off, the plane requires all engines. Once cruising altitude is reached, the plane can survive with only three out of

four engines. Landing only requires two of those engines to be operational. In this scenario, the "k out of n" requirements for the reliability block diagram changes during the simulation based on the passage of time during the flight. This idea is being pursued by the Air Force Operational Test and Evaluation Center in their changes to RAPTOR. They consider this approach a phased simulation model and are working to incorporate parameter changes to reliability block diagrams based on time.

Conclusions

Advancements to system-level reliability analysis are provided in this research. Current issues facing reliability engineers are explored and suggestions are made to improve their analysis. The lack of failure data is handled by using the triangle distribution to gain accurate system performance metrics. Sparing policies are compared based on availability and cost criteria to assist decision-makers in choosing from alternative maintenance strategies. The confusion among reliability engineers dealing with time to failure and time between failure distribution is highlighted and an example is provided to further clarify the correct and incorrect interpretation of these results.

The understanding and improvement in RM&A analysis is augmented by improvements to state-of-the-art simulation software. The RAPTOR simulation software is provided in the public domain and these modifications will be presented to the developing office for consideration in future software changes. Many of these improvements will be included or modified as additional features in future software releases. This research benefits the Air Force and the industrial community as reliability simulation continues to grow and gain credibility in the reliability literature.

The doors to continued research are opened and ideas are presented for improving on the groundwork of this dissertation. The individual case studies can be expanded along with developing new scenarios to apply this analysis. Increasing the degree to which a real-world system can be modeled should be the goal of future research. The better the system can be modeled, the better the system can be explained. This clarify RM&A analysis and increasing the understanding of system behavior in the reliability community.

REFERENCES

Air Force Operational Test and Evaluation Center (1995). *Rapid Availability Prototyping for Testing Operational Readiness (RAPTOR)*, User's Manual, Albuquerque, NM.

Ascher, H. and Feingold, H. (1984). *Repairable Systems Reliability: Modeling, Inference, Misconceptions and Their Causes*, Lecture Notes in Statistics, Volume 7, Marcel Dekker, Inc., New York, NY.

Barlow, R.E. and Proschan, F. (1965). *Mathematical Theory of Reliability*, John Wiley & Sons, New York, NY.

Banks, Jerry, Carson II, John S., and Nelson, B. L. (1996). *Discrete-Event System Simulation*, Prentice-Hall Inc., Upper Saddle River, NJ.

Billington, R. and Allan, R. N. (1992). *Reliability Evaluation of Engineering Systems, Conepts and Techniques*, 2nd Edition, Plenum Press, New York, NY.

Blackstone, J. H., (1979). "Reliability Simulation", Doctoral Dissertation, Texas A&M.

Best Fit Software (1994), Version 1.11, Palisade Corporation.

Burns, Deborah Jane. (1994). "A Macro Language Computer Simulation Tool for Series System Reliability Analyses", Master's Thesis, Arizona State University.

CACI Products (1995). *MODSIM II: The Language for Object-Oriented Programming*, User's Manual, La Jolla, CA.

Conway, A. E., and Goyal, A. (1987). "Monte Carlo Simulation of Computer System Availability/Reliability Models", *Proceedings 17th Annual Symposium on Fault-Tolerant Computing*, pp. 230-235.

Design-Expert 5, Version 5.0.8, Stat-Ease Corporation, Minneapolis, MN.

Durkee, Darren P.; Pohl, Edward A.; and Mykytka, Edward F. (1997). "Sensitivity of Availability Estimates to Input Data Characterization", *Submission for the Third Annual ISSAT International Conference on Reliability and Quality in Design.*

Elsayed, E. A. (1996). *Reliability Engineering*, Addison Wesley Longman Inc., Reading, MA.

Foster III, Joseph W.; Hogg, Gary L.; and Gonzales-Vega, Ofelia (1986). "Simulation Methodology of Large Scale Systems", *Proceeding Annual Reliability and Maintainability Symposium*, pp. 419-426.

Glynn, P. W. (1989). "A GSMP Formalism for Discrete Event Simulation", *Proceedings, IEEE*, Vol. 77, No. 1, pp. 14-23.

Gonzales-Vega, Ofelia (1987). "Reliability Simulation", Doctoral Dissertation, Texas A&M.

Gonzales-Vega, Ofelia, Foster III, Joseph W., and Hogg, Gary L. (1988). "A Simulation Program to Model Effects of Logistics on R&M of Complex Systems", *Proceeding Annual Reliability and Maintainability Symposium*, pp. 306-313.

Gopalakrishnan, K. (1985). Discrete Event Reliability Maintainability Availability (DERMA) Model, *TIMS/ORSA Joint National Meeting*, April.

Gross, Donald, and Harris, Carl M., (1985). "*Fundamentals of Queueing Theory*", 2nd Edition, Wiley Series in Probability and Mathematical Statistics, John Wiley & Sons, New York, NY.

Hassett, Thomas F., Dietrich, Duane L., and Szidarovszky, Ferenc (1995). "Time-Varying Failure Rates in the Availability & Reliability of Repairable Systems", *IEEE Transactions on Reliability*, Vol. 44, No. 1, March, pp. 155-160.

Hoffman, David J., and Viterna, Larry A., (1991). *ETARA PC Version 3.3, Reliability, Availability, Maintainability Simulation Model,* User's Guide, NASA Lewis Research Center, Cleveland, OH.

Hwang, C. I.; Tillman, F. A.; and Lee, M. H. (1981). "System-Reliability Evaluation Techniques for Complex-Large Systems – A Review", *IEEE Transactions on Reliability*, Vol. R-30, Issue 5, pp. 117-125.

Jossely, J. V., Fleming, R. E., Frenster, J. A., and De Hoff, R. L. (1986). "Application of Markov Models for RMA Assessment", *Proceedings Annual Reliability and Maintainability Symposium*, pp. 427-432.

Kales, Paul (1998). *Reliability for Technology, Engineering, and Management*, Prentice Hall, Upper Saddle River, NJ.

Kapur, K. C., and Lamberson, L. R. (1977). *Reliability in Engineering Design*, John Wiley & Sons, City

Kececioglu, Dimitri, (1991). *Reliability Engineering Handbook*, Vol. 1 and 2, Prentice-Hall Inc., Englewood Cliffs, NJ.

Kececioglu, Dimitri, (1991). *Reliability Engineering Handbook*, Vol. 1 and 2, Prentice-Hall Inc., Englewood Cliffs, NJ.

Kovalenko, I. N.; Kuznetsov, N. Y.; and Pegg, P. A. (1997). *Mathematical Theory of Reliability of Time Dependent Systems with Product Application*, Wiley Series in Probability and Statistics, Chichester, England.

Landers, Thomas L., Taha, Hamdy A., and King, Charles L. (1991). "A Reliability Simulation Approach for Use in the Design Process", *IEEE Transactions on Reliability*, Vol. 40, No. 2, June, pp. 177-181.

Law, Averill M., and Kelton, David W. (1982). *Simulation Modeling and Analysis*, 2nd Edition, McGraw-Hill Inc., New York, NY.

Leitch, R. D. (1995). *Reliability Analsyis for Engineers, An Introduction*, Oxford University Press, Oxford.

Mackulak, G. T., and Cochran, J. K. (1990). "Generic/Specific Modeling: An Improvement to CIM Simulation Techniques", *Optimization of Manufacturing Systems Design*, pp. 237-259.

Mackulak, G. T.; Savory, P. A.; and Cochran, J. K. (1994). "Ascertaining Important Features for Industrial Simulation Environments", *Simulation*, Vol. 63:4, pp. 211-221.

Minitab, Release 11.13, Minitab Incorporated.

Moss, M. A. (1996). *Applying TQM to Product Design and Development*, Quality and Reliability Series, Marcel Dekker Inc., North Hills, CA.

Mongomery, D. C. (1991). *Design and Analysis of Experiments*, 3rd Edition, John Wiley & Sons, New York, NY.

Ozdemirel, N. E., and Mackulak, G. T. (1993). "A Generic Simulation Module Architecture Based on Clustering Group Technology Model Codings", *Simulation*, Vol. 60:6, pp. 421-433.

Pristsker, A. Alan B. (1986). *Introduction to Simulation and SLAM II*, 3rd Edition, John Wiley & Sons, New York, NY.

Stalnaker, D. K. (1993). *ACARA,* User's Guide, NASA Lewis Research Center, Cleveland, OH.

Usher, J. S. (1993). "Case Study: Reliability Models and Misconceptions", *Quality Engineering*, Vol. 6, pp. 261-271.

Villemeur, Alain (1992). *Reliability, Availability, Maintainability and Safety Assessment, Methods and Techniques*, Vol. 1, John Wiley & Sons, Chichester, England.

Windebank, E. (1983). "A Monte Carlo Simulation Method Versus a General Analytical Method for Determining Reliability Measures of Repairable Systems", *Reliability Engineering*, Vol. 5, pp. 73-81.

Zacks, Shelemyahu (1992). *Introduction to Reliability Analysis, Probability Models and Statistical Methods*, Springer-Verlang, New York, NY.

BIOGRAPHICAL SKETCH

Captain Stephen Paul Chambal was born in Flint, Michigan, on the 13th of November 1968. He received his elementary education at Millington Elementary School in Millington, MI. His secondary education was completed at Millington High School in Millington, MI in 1986. After high school he enlisted in the United States Air Force and worked in the maintenance career field before accepting an appointment to the Air Force Academy. He was a distinguished graduate in 1993 with Bachelor of Science degrees in Mathematical Sciences and Operations Research and received a regular commission. He attended Arizona State University immediately after graduation and earned his Master of Science Degree in Industrial Engineering in December of 1994. His next assignment took him to Albuquerque, New Mexico as a logistics analyst for the Headquarters Air Force Operational Test and Evaluation Center. In August 1996 Captain Chambal returned to the Graduate College at Arizona State University to pursue a doctorate in Industrial Engineering with an emphasis on quality and reliability engineering. Following graduation, Captain Chambal will take a position as a professor at the Air Force Institute of Technology in Dayton, OH. Captain Chambal is married to the former Rhonda Kay McCullah of Vandalia, OH. They have no children, as of yet.